UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

Librería de bandidos multi-brazo para recomendación

Autor: David Cabornero Pascual
Tutor: Pablo Castells Azpilicueta

mayo 2021

David Cabornero Pascual
Librería de bandidos multi-brazo para recomendación

David Cabornero Pascual

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

- La ilusión no se come - dijo ella.

- No se come, pero alimenta - dijo el coronel.

Gabriel García Márquez

Agradecimientos

Con este trabajo acabo una de las etapas más importantes de mi vida: la de estudiante. En estos cinco años ha habido momentos dulces y amargos, pero no los cambiaría por nada en el mundo. En primer lugar, muchas gracias a mis padres por el cariño, la confianza y el mecenazgo, y en especial a mi abuela por acogerme en su casa estos años. Por supuesto, gracias a Juanmi y a Laura por el apoyo, las risas y por aguantarme cuando el estrés me volvía insoportable en épocas de exámenes. No os hacéis a la idea de lo terapéuticas que han sido nuestras charlas.

Finalmente, estoy obligado a agradecer a Mario, Alejandro, Miguel, Sergio y Mané su mera presencia durante estos cinco años. Objetivamente, mi rendimiento académico no hubiera sido el mismo sin vosotros, pero además me habéis hecho mejor persona, y si he evolucionado estos años ha sido en gran parte gracias a vosotros. No hay suficientes pizarras en el mundo para que pueda plasmar lo que he aprendido de vosotros y con vosotros.

Muchas gracias a todos, sois los mejores.

Resumen

En las últimas décadas, los algoritmos de recomendación han cobrado vital importancia. Plataformas como Netflix, Facebook, Twitter o Amazon se han incrustado en nuestras vidas y ofertan miles (o millones) de opciones posibles. Estos algoritmos ayudan a seleccionar los *items* más interesantes para el cliente, mejorando así la experiencia del usuario y los beneficios de las empresas.

La perspectiva tradicional sugiere que, conocido un cierto conjunto de datos, se ofrezca un artículo o *item* al usuario y se acabe ahí el proceso de recomendación. En cambio, el Aprendizaje Reforzado ofrece una perspectiva distinta respecto a estos algoritmos, ya que entiende la acción de recomendar como un proceso iterativo en el que se puede aprender de la respuesta del usuario.

Dentro del Aprendizaje Reforzado, los bandidos multi-brazo son la rama más fructífera en lo que se refiere a sistemas de recomendación. En ella, se tiene un conjunto de acciones entre las que el algoritmo tiene que elegir en cada unidad de tiempo. A su vez, cada acción sigue una estrategia distinta para recomendar un cierto *item*.

El objetivo de este proyecto es crear una librería funcional que recopile los algoritmos más importantes de esta rama, permitiendo que en un futuro haya invesigadores que la mejoren, amplíen o simplemente trabajen con ella. En concreto, se van a implementar 9 algoritmos: ε-greedy, UCB, CNAME, Thompson Sampling, EXP3, Gradiente, LinUCB, DynUCB y CLUB. Además de la propia librería, se experimentará con ella y se mostrarán resultados que muestren su calidad y variedad de recomendación.

Palabras clave

Aprendizaje reforzado, bandidos multi-brazo, sistema de recomendación, acción, brazo, recompensa, *recall* acumulado.

Abstract

During the last decades, recommender systems have become highly relevant. Online platforms like Netflix, Facebook, Twitter or Amazon have took part in our lives and they offer thousands (or millions) of possibilities. These algorithms help us to select the most interesting items for the client, enhancing user's experience and company's profits.

The traditional point of view suggests that, once known a certain dataset, an item is offered to the user. When the item is recommended, the recommendation proccess ends. On the other hand, Reinforcement Learning tecniques offer a disinct perspective, since they understand the action of recommending as an iterative process which they can learn about the user's feedback.

Respect to Reinforcement Learning, multi-armed bandits are the most fructiferous branch if we are working with recommender systems. On this part, we have a set of actions among which the algorithm has to choose in each epoch. At the same time, each action follows a different strategy to recommend items.

The objective of this project is to create a library that collects the most important algorithm of this branch, allowing researchers to improve it, extend it or work with it. Specifically, 9 algorithms will be implemented: ε-greedy, UCB, CNAME, Thompson Sampling, EXP3, Gradient, LinUCB, DynUCB and CLUB. In addition to the library itself, experiments and results that shows the quality and variety of recommendations will be included.

Keywords

Reinforcement Learning, Multi-armed bandits, Recommender Systems, action, arm, reward, cumulative recall.

ÍNDICE

Listas

Lista de algoritmos

Lista de ecuaciones

Lista de figuras

Lista de tablas

1

Introducción

Los sistemas de recomendación están cada vez más presentes en nuestras vidas debido al exceso de oferta que nos ofrecen muchas plataformas: miles de productos en páginas de compra, una infinidad de *posts* en nuestras redes sociales, millones de vídeos en plataformas como *YouTube*, miles de películas en plataformas de *streaming*... Por nuestro bien y el de estos sitios, se debe establecer una preferencia entre todos los artículos y mostrar los más relevantes.

El paradigma de estos algoritmos normalmente consiste en obtener toda la información posible de los usuarios y explotarla para optimizar sus recomendaciones. Sin embargo, en el paradigma de aprendizaje reforzado se sigue un modelo ensayo-error en el que el propio algoritmo aprende qué estrategia es mejor seguir, evolucionando así conforme obtiene más información. Esto permite a los sistemas de recomendación empezar de cero en vez de necesitar una gran cantidad de datos para ser capaces de recomendar.

En este trabajo se ha implementado una librería con nueve algoritmos multi-brazo (principal rama del aprendizaje reforzado) y las herramientas necesarias para investigar su eficiencia. [1] Aunque es un trabajo que a priori podría considerarse tecnológico por consistir en la implementación de una librería, la novedad de toda la algoritmia que hay detrás ha motivado a experimentar con la librería y mostrar resultados básicos. Por ello, se considera que se ha seguido una metodología científica durante el desarrollo e investigación del proyecto.

1.1. Motivación

El aprendizaje reforzado es una rama poco trabajada dentro del aprendizaje automático. Si además nos referimos en concreto a los bandidos multi-brazo y los aplicamos a los sistemas de recomendación, la literatura es muy escasa para los buenos resultados que se encuentran en ella. Aunque se encuentren numerosos artículos con algoritmia muy variada y novedosa, no parece que nadie se haya decidido a recopilar la mayoría bajo una misma librería. En este sentido, la novedad que aporta este trabajo es una nueva librería con la que más personas puedan conocer e investigar sobre estos

[1] La librería puede encontrarse en https://github.com/DCabornero/TFGReinforcementLearning.

algoritmos.

La segunda motivación de este trabajo consiste en trabajar y mostrar algunos algoritmos contextuales, ya que no han sido tratados aún en esta escuela y existe muy pocos artículos al respecto. Por lo tanto, con esto se pretende abrir paso y dar un poco de luz a una rama del aprendizaje reforzado que ni tiene mucha edad ni ha sido explorada en profundidad.

Por último, también se pretende dar una descripción teórica de todos los algoritmos para dar una intuición al lector sobre cómo funcionan y bajo qué principios se rigen. Por ello, aparte de la propia librería se quiere dejar una constancia de algunos fundamentos teóricos de la rama sin llegar a un nivel demasiado técnico e inaccesible.

1.2. Objetivos

Con la motivación anteriormente expuesta, se muestran a continuación los objetivos básicos de este trabajo:

- Diseñar una librería extensa y completa que pueda utilizarse para labores de investigación en la rama de bandidos multi-brazo en sistemas de recomendación.
- Unificar numerosas implementaciones que se encontraban desperdigadas por la Web en numerosos *papers*.
- Proporcionar herramientas para analizar y optimizar algoritmos en cuestión.
- Dar al usuario la oportunidad de probar diversas métricas para medir distintos aspectos de las recomendaciones obtenidas.
- Mostrar un análisis detallado de la algoritmia aplicada a varias bases de datos.
- Aportar al mundo una librería flexible que pueda extenderse y modificarse si ningún tipo de problema.

1.3. Estructura del documento

El documento en cuestión está dividido en las partes que se enumeran:

- **Estado del arte**. Repasa los conceptos básicos de los sistemas de recomendación, el aprendizaje reforzado y los bandidos multi-brazo, necesarios para entender el trabajo.
- **Diseño**. Muestra cómo se ha estructurado el proyecto, la metodología seguida y la notación que se va a utilizar durante los siguientes capítulos.

- **Algoritmia**. Junto con el siguiente capítulo, trata el tema central del proyecto. En él, se describen todos los algoritmos implementados en la librería. Su principal función es crear una intuición en el lector sobre cómo funciona cada uno.
- **Experimentos y resultados**. Mediante búsqueda en rejilla se buscan los mejores hiperparámetros para cada algoritmo. Después, se comparan todos con distintas métricas típicas de sistemas de recomendación.
- **Conclusiones**. Se recapitula todo lo visto en en este trabajo, se sintetizan sus principales aspectos y se proponen nuevas ramas de investigación que hayan podido quedar abiertas.

2

Estado del arte

2.1. Sistemas de recomendación

Durante estas últimas décadas, los sistemas de recomendación han empezado a tener una importancia en nuestras vidas debido a plataformas como las redes sociales o los servicios de contenido por *streaming*. Este crecimiento ha venido acompañado de una gran gama de nuevos algoritmos y paradigmas en esta disciplina [Aggarwal, 2016], pero en términos generales los datos con los que se trabajan son:

- Un conjunto $\mathcal{U}$ de **usuarios** que utilizan el sistema.
- Un conjunto $\mathcal{I}$ de ***items*** con los que los usuarios interaccionan.
- Un conjunto $\mathcal{R}$ de valoraciones o ***ratings***, donde cada elemento viene dada por la tupla (u, i, r), donde $u \in \mathcal{U}$, $i \in \mathcal{I}$ y r es la valoración que ha dado el usuario u al item i.
- Un contexto opcional. Puede ser sobre las características de los usuarios, de los *items* o cualquier otro tipo de información.

Las valoraciones pueden ser de muchos tipos: numéricas en un intervalo, binarias, implícitas porque el usuario no suele valorar... Además, lo normal es que un usuario en concreto no haya valorado un cierto *item*. Por lo tanto, si hiciéramos una matriz de valoraciones de tamaño $|\mathcal{U}| \times |\mathcal{I}|$, donde las filas representan los usuarios y las columnas los *items*, casi toda la matriz estaría vacía.

De aquí surgen dos formas de entender el objetivo final de un sistema de recomendación:

- **Versión predictiva**: el objetivo final es rellenar la matriz de valoraciones con los datos existentes de la manera más precisa posible.
- **Versión del *ranking***: en muchas ocasiones no se necesita saber con precisión qué opinaría el usuario de cada *item*, sino que es suficiente saber los *items* con los que guarda más afinidad para que sean esos los que se recomienden. Por lo tanto, en este caso solo interesa conocer el *top del ranking*.

Depende de la información que se disponga a la hora de recomendar un *item*, existe la siguiente clasificación de algoritmos [Aggarwal, 2016, capítulo 1]:

- **Filtrado colaborativo**. Solo utiliza los hábitos de consumo de los usuarios y sus similitudes. No utiliza las características propias de los *items*.
- **Algoritmos basados en el contenido**. Solo se utilizan las características y el contenido de los *items* que se van a recomendar. Por lo tanto, aquí se trata a los usuarios de forma completamente independiente y no se aprovechan las valoraciones de otros usuarios.
- **Modelos híbridos**. Los dos modelos anteriores no son complementarios, sino que pueden aprovecharse a la vez las relaciones con otros usuarios y las características de los *items*.

Por otro lado, los métodos de evaluación son un área abierta muy importante en sistemas de recomendación, ya que dependen completamente de la situación en que nos encontremos. Aunque hay más tipos de evaluación, una forma genérica de clasificarlos [Vargas, 2014, capítulo 2.2] es mediante evaluación *online* y *offline*:

- **Evaluación *offline* o estacionaria**. En este tipo de sistemas primero se obtiene una cierta información y después se evalúa el resultado. Por lo tanto, la base de datos que se utiliza al principio permanece estática durante todo el proceso de aprendizaje. Esta forma de evaluación suele ser más simple y eficiente.

  Normalmente, se separa el conjunto de *ratings* $\mathcal{R}$ en $\mathcal{R}_{train}$ y $\mathcal{R}_{test}$, de forma que se entrena el algoritmo con el primero mientras que segundo se utiliza para contrastar las predicciones o el ranking que se hayan obtenido.
- **Evaluación *online***. El sistema va recibiendo más datos conforme pasa el tiempo, por lo que los conjuntos anteriores son cada vez mayores. Es un sistema más típico de empresas e instituciones que quieren desviar una pequeña parte de sus datos a una plataforma de recomendación para entrenarla.

2.2. Aprendizaje Reforzado

El Aprendizaje Reforzado [Sutton and Barto, 2018, capítulo 1] es una disciplina del *Machine Learning* en la que se dispone de un conjunto de acciones entre las que el algoritmo (agente) debe decidir en cada momento. Cada vez que el algoritmo escoge una acción, recibe una recompensa del entorno acompañada de su estado actual. De esta forma, el agente va escogiendo opciones que maximicen su recompensa. Este paradigma del Aprendizaje Automático, como puede verse, escapa de los términos tradicionales de aprendizaje supervisado y no supervisado.

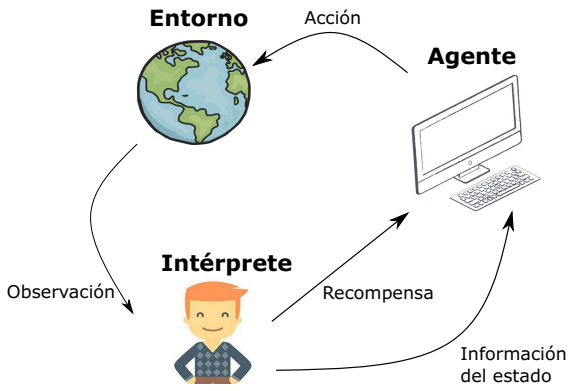


Figura 2.1: Esquema del funcionamiento de un algoritmo de Aprendizaje Reforzado.

El caso más complejo de estos algoritmos viene cuando una acción no solo influye en la recompensa obtenida, sino que también influye en las recompensas posteriores (por ejemplo, en juegos). Estas dos características (búsqueda de ensayo-error y recompensas con retraso) son las dos cualidades que distinguen al Aprendizaje Reforzado de otras ramas del aprendizaje automático.

Además, este tipo de algoritmos son especialmente **flexibles**, ya que si se les expone a situaciones para las que no han sido entrenados se espera que sean capaces de adaptarse y encontrar buenas soluciones para este nuevo tipo de problemas.

Una de las principales complicaciones que se encuentran en estos algoritmos en especial es lo difícil que es compensar la **exploración** y la **explotación**. Por un lado, el algoritmo debe explorar todas las acciones para tratar de descubrir cuáles son aquellas con las que puede obtener mayores recompensas. Sin embargo, también debe explotar aquellas opciones que considere mejores para obtener buenas recompensas a corto plazo.

El dilema exploración-explotación es en general un problema abierto que sigue siendo estudiado, por lo que no existe una forma sencilla de hallar una proporción adecuada ni de saber cuándo se debe explorar y cuándo explotar. Sin embargo, es evidente que al principio es necesario maximizar la exploración para conseguir información de manera masiva, mientras que una vez que se ha conseguido suficiente información se puede aumentar la explotación. Al fenómeno que trata cómo los algoritmos deben gestionar un comienzo con poca información se le llama **arranque en frío**. El Aprendizaje Reforzado es especialmente bueno a la hora de empezar en frío, ya que no necesita un conjunto de entrenamiento para empezar a realizar predicciones y aprender de su *feedback*.

Las dos ramas más importantes dentro del Aprendizaje Reforzado son los **bandidos multi-brazo** y los **procesos finitos de decisión de Markov**. En este trabajo hemos decidido centrarnos en los primeros, ya que existe una literatura mucho más rica en lo relativo a los sistemas de recomendación.

2.3. Bandidos multi-brazo

Este nombre viene de una curiosa traducción al español de *multi-armed bandit*, ya que en inglés significa *máquina tragaperras*. En este tipo de problemas [Sutton and Barto, 2018, capítulo 2], tenemos que elegir un cierto número de veces (llamado épocas) una cierta acción a (también llamada brazo) del conjunto de acciones $\mathcal{A}$. Tras escoger una cierta acción, la recompensa viene dada por un valor numérico **estacionario**. El objetivo, por supuesto, es maximizar estas recompensas.

En una cierta época, el algoritmo estima una cierta recompensa para cada una de las acciones. Habitualmente, se llama $Q_t(a)$ a la estimación de una cierta acción $a \in \mathcal{A}$, mientras que a la recompensa esperada teórica se la denomina $q(a)$. Al estar tratando con problemas estacionarios, $q(a)$ es un valor constante que el algoritmo debe averiguar con la mayor precisión posible en el transcurso de las iteraciones.

Pongamos que, tras una serie de iteraciones, el algoritmo ha estimado una recompensa $Q_t(a)$ para cada acción $a \in \mathcal{A}$. Un algoritmo que escogiera la opción con mayor recompensa estimada (opción *codiciosa*) se espera que consiga la máxima recompensa en la siguiente iteración. Es a esto a lo que llamamos explotación en los sistemas multi-brazo.

Sin embargo, no podemos realizar este mecanismo siempre, ya que es necesario seguir ganando información sobre el resto de acciones para ver si realmente existe una acción mejor. Además, en sistemas de recomendación cabe la posibilidad de que la mejor acción cambie por el tiempo. Por todo esto, es necesario que nuestro algoritmo escoja acciones subóptimas para mejorar la estimación de recompensa de todos los brazos. Es a esto a lo que llamamos exploración en los sistemas multi-brazo.

Volvamos al problema que nos ocupa, **la aplicación de bandidos multi-brazo a sistemas de recomendación**. En un sistema de recomendación, recibimos un usuario $u \in \mathcal{U}$ al que debemos recomendar un *item* que consiga una buena valoración. Claramente, cada acción tiene que seleccionar un *item* y conseguir una cierta recompensa, que será la valoración del usuario (Figura 2.2). Las recompensas están claras, ¿pero cuáles son las acciones?

La opción más intuitiva y sencilla es que **cada acción escoja un cierto *item***, de forma que haya una correspondencia biyectiva entre el conjunto de *items* $\mathcal{I}$ y el conjunto de acciones $\mathcal{A}$. Este va a ser el método a seguir durante nuestro trabajo, por lo que en lo que sigue se hablará de brazos e *items* indiferentemente y nos referiremos al conjunto de *items* también como $\mathcal{A}$. En una versión más elaborada, también puede estimarse la recompensa no solo en función del brazo escogido, sino del contexto que exista en ese momento.

Sin embargo, existen otras formas de escoger las acciones, cambiando el paradigma habitual. Un ejemplo puede ser que los brazos no sean *items* sino usuarios, de forma que el brazo escogido sea un vecino similar que nos sirva para elegir un *item* recomendado [Sanz-Cruzado et al., 2019]. Otra forma posible es que los *brazos* sean distintos algoritmos de recomendación, haciendo así que nuestro

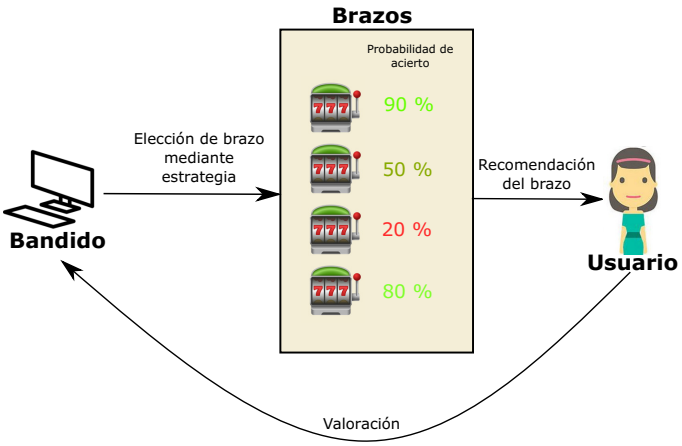


Figura 2.2: Esquema básico de un bandido multi-brazo en recomendación.

algoritmo multi-brazo actúe como un *ensemble* [Cañamares et al., 2019]. En cualquier caso, durante todo este trabajo será equivalente hablar de una acción o un *item*.

Aunque los bandidos multi-brazo están destinados a problemas estacionarios, no es habitual que este sea el caso en sistemas de recomendación, ya que suelen estar muy ligados a la fugacidad y la temporalidad. Sin embargo, algunos algoritmos son capaces de adaptarse satisfactoriamente a perturbaciones no demasiado grandes y serán definidos en los siguientes apartados.

2.4. Métricas de evaluación en recomendación interactiva

En los sistemas de recomendación interactivos (como cuando se aplican *bandits*) no hay una única heurística que haya que explotar para maximizar los buenos resultados. Aparte de hacer buenas recomendaciones hay que tener en cuenta otros factores más a largo plazo, ya que queremos que los usuarios permanezcan en la plataforma después de mucho tiempo. Es por ello que en las recomendaciones se tienen en cuenta otros factores como la diversidad, la novedad o que el algoritmo sorprenda de vez en cuando al usuario [Aggarwal, 2016, capítulo 1].

Por todo esto, es importante disponer de varias métricas que nos permitan determinar que el algoritmo cumple varios de los requisitos anteriormente mencionados. A continuación se muestran las dos más importantes, que serán clave a lo largo del trabajo:

- ***Recall* acumulado**: es la métrica de sistemas de recomendación por antonomasia. Consiste en calcular la proporción de valoraciones positivas que el algoritmo ha sido capaz de descubrir tras un cierto tiempo. Si nos fijamos en la evolución del algoritmo a lo largo del tiempo, este valor siempre empieza en 0 y es no decreciente, alcanzando como máximo el 1.

- **Coeficiente de Gini** [StatsDirect, 2013]: sirve para medir la variedad de recomendaciones [1]. Toma un valor entre 0 y 1, donde 0 significa que las recomendaciones están distribuidas equitativamente entre los *items*.

  A continuación se define dicho coeficiente. Sea i un cierto *item* del conjunto de *items* disponibles $\mathcal{I}$, sea x_i el número de veces que se ha recomendado dicho *item* y sea $n > 0$ el número total de *items*. Entonces, el coeficiente de Gini es:

$$G = \frac{\sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{I}} |x_i - x_j|}{2n \sum_{i \in \mathcal{I}} x_i}. \tag{2.1}$$

  Supongamos además que tenemos el conjunto $\{x_1 \dots x_n\}$ está ordenado de forma ascendente. Entonces, el algoritmo pasa de tener complejidad $O(n^2)$ a complejidad $O(n)$:

$$G = \frac{\sum_{i=1}^{n} (2i - n - 1) x_i}{n \sum_{i=1}^{n} x_i}. \tag{2.2}$$

  Sin embargo, si comenzamos con el conjunto de datos no ordenados podemos ordenarlo y después utilizar el segundo algoritmo. En tal caso, el algoritmo tiene complejidad $O(n \log n)$.

[1] También se usa en otros campos como la economía, donde se usa como baremo para medir la desigualdad entre los ingresos de los ciudadanos.

3

Desarrollo

3.1. Metodología

La metodología que se ha seguido para cumplir con los objetivos del trabajo ha sido la siguiente:

1. Como estamos en un trabajo de investigación, la primera parte ha consistido en un estudio del aprendizaje reforzado y los sistemas de recomendación en general. Posteriormente, se han completado estos conocimientos con una investigación sobre algoritmos propios de la rama que nos ocupa.
2. Una vez conocidos los algoritmos, se ha implementado una librería y se ha comprobado su correcto funcionamiento.
3. Cuando se ha finalizado la librería, se ha abierto una fase de experimentación en la que se comprueba cómo funcionan los algoritmos con distintas bases de datos.
4. Una vez dominados los algoritmos y conocidos sus resultados en implementaciones reales, se procede a crear en esta memoria una documentación técnica de cada uno. En ella, se muestran de manera sencilla los principios que subyacen tras cada algoritmo.

Por la propia naturaleza de la investigación, la primera y segunda fase se iteran una y otra vez hasta tener formada la librería. Una vez formada la librería en sí es cuando se han realizado los experimentos y la documentación correspondiente. [1].

3.2. Parámetros configurables de la librería

Como toda librería, se permiten modificar parámetros para flexibilizar las ejecuciones posibles. En este apartado se muestran todos los parámetros que se pueden configurar en la ejecución, así como aquellos que se han mantenido rígidos por algún propósito. Comencemos con los parámetros que se pueden elegir:

[1] Dicha librería puede encontrarse en https://github.com/DCabornero/TFGReinforcementLearning

- **Selección de usuario**. En cada época se elige un usuario distinto para hacerle una recomendación. En esta librería se ofrecen tres métodos de elección de usuario:
  - *Random no balanceado*. En él, cada usuario es elegido completamente al azar en cada época.
  - *Random balanceado*. En él, se sortea la lista de usuarios para determinar el orden en que serán elegidos. Una vez se ha acabado la lista, se vuelve a sortear para repetir este proceso tantas veces como sea necesario. De esta forma, las recomendaciones están equilibradas entre los usuarios.
  - *Round-robin*. La lista de usuarios se sortea una única vez, y cuando se ha acabado se vuelve a empezar. De esta forma, no hay un sorteo aleatorio, sino que siempre se sigue el mismo orden de recomendación.
- **Recompensa del *miss***. Cuando no se encuentra un dato, se puede decidir qué recompensa se da al sistema. En sistemas de recomendación es muy habitual que la ausencia de recomendación suponga *a priori* una mala valoración, ya que se suele asumir que existen más artículos que no le gustan al usuario. Por ello, se recomienda dejar este valor a 0, aunque se puede escoger cualquier valor entre 0 y 1.
- **Permanecer en la época si hay *miss***. Aunque la práctica habitual es no hacerlo, puede interesar contar solo aciertos y fallos. Por ello, se propone una alternativa en la que no se avanza de época cuando no encuentras un dato.
- **Métricas**. Puede interesar guardar datos sobre la evolución de ciertas métricas según pasan las épocas. Por ello, se permite opcionalmente guardar el histórico de la evolución del tiempo de ejecución y del coeficiente de Gini. En concreto, hay que tener en cuenta que el cálculo del coeficiente de Gini es $O(n \log n)$ y que tarda un tiempo razonable. Por ello, el tiempo de ejecución se verá alterado si se pide calcular también el histórico del coeficiente de Gini.

Además, de esto, todos cada algoritmo *bandit* tiene sus propios parámetros, y se detallarán en el capítulo siguiente. Veamos ahora los parámetros que se han dejado rígidos y por qué se ha decidido esto:

- ***Recall* acumulado**. Se considera la métrica central, por lo que siempre se guarda su evolución según pasan las épocas. Hay que destacar que, a diferencia del coeficiente de Gini, el cálculo del *recall* acumulado no causa retrasos en la ejecución.
- **No se vuelve a recomendar un mismo *item* a un mismo usuario**. Esto se ha decidido para dar más sentido al coeficiente de Gini y al *recall* acumulado, haciendo así que el segundo sea proporcional al conjunto de aciertos. Además, es una de las técnicas más sencillas que evitan la sobreespecialización en unos pocos *items* por parte de los algoritmos.

- **Recompensas binarias**. Aunque la ausencia de dato sí tiene una valoración que puede cambiarse, no ocurre esto con las valoraciones positivas y negativas, que tienen recompensa 1 y 0, respectivamente. Esto se ha decidido para dar flexibilidad a la elección de base de datos que vaya a utilizar y para simplificar algunos algoritmos (como el gradiente).

3.3. Arquitectura e implementación de la librería

La arquitectura de la librería va a seguir un esquema orientado a objetos, donde el objeto principal es *Bandit*, una clase abstracta de la que partirán todos los algoritmos implementados. El diagrama de clases de esta arquitectura se muestra en la Figura 3.1.

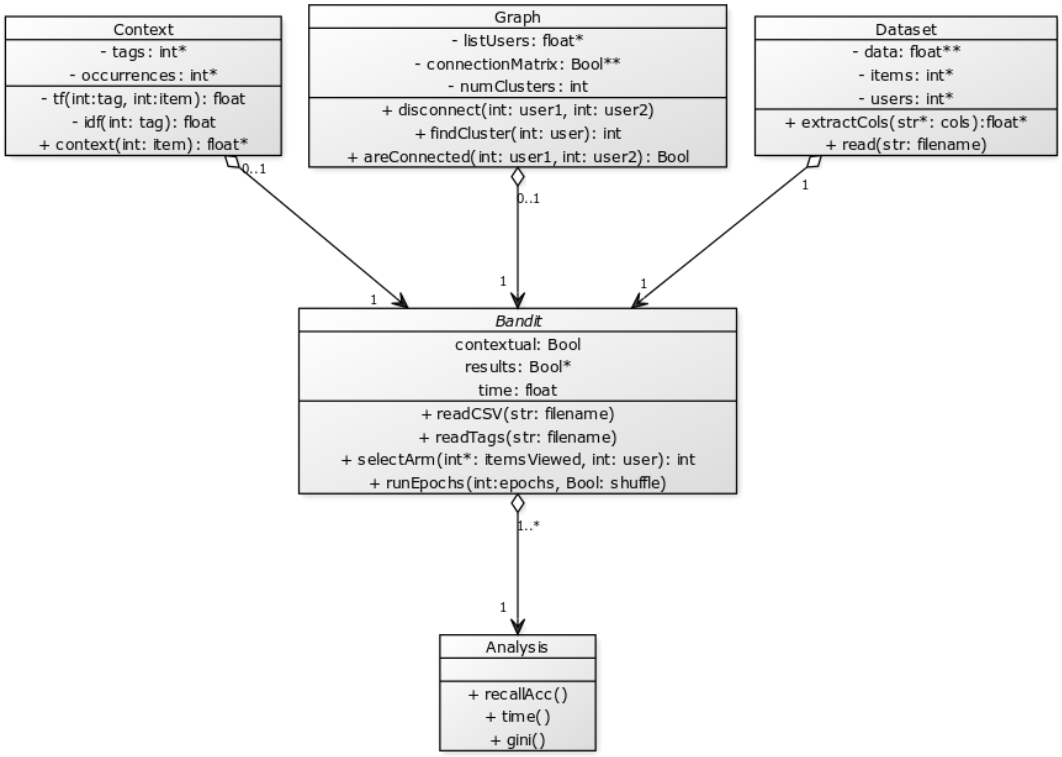


Figura 3.1: Diagrama de clases de la librería.

A continuación se describen todas las clases y se especifica la función de cada una:

- *Dataset*. Clase encargada de organizar los datos de archivos de valoraciones. Permite acceder fácilmente a las columnas que se deseen.
- *Context*. Solo necesitan esta clase los bandidos contextuales. En ella, cada *item* tiene unos ciertos *tags* asociados, incluyendo un *tag* obligatorio *no-tag*, necesario para que los algoritmos funcionen correctamente. El cometido de esta clase es calcular un vector contexto para cada clase mediante el algoritmo *Tf-idf* (más información en Apéndice A).

- *Graph*. Solo es necesario para el algoritmo contextual *CLUB*. Se encarga de guardar un grafo con todos los usuarios que inicialmente es completo, es decir, todos los nodos están conectados con todos los nodos. Según pasen las épocas se irán desconectando aristas del grafo y la clase debe de ser capaz de responder qué usuarios están en cada componente conexa.
- *Bandit*. Es el centro neurálgico de la librería: todos los algoritmos heredan de esta clase. Debe de guardar el historial de las recomendaciones exitosas, las fallidas y el tiempo de ejecución. A continuación se nombran los principales métodos expuestos en el diagrama de la Figura 3.1:
  - *readCSV* y *readTags* se encargan de leer los ficheros CSV de valoraciones y de *tags* para el contexto.
  - *selectArm* se encarga de recomendar a un cierto usuario un cierto *item* con la información de la que dispone. *itemsViewed* son los *items* ya vistos, ya que las recomendaciones no pueden repetirse.
  - *runEpochs* realiza un cierto número de recomendaciones seguidas barajando a los usuarios. El sistema debe recomendar a todos los usuarios un número similar de *items*.
- *Analysis*. Esta clase es la encargada de analizar varias clases *Bandit*, comparando resultados como el tiempo de ejecución, el *recall* acumulado o el coeficiente de Gini.

3.4. Ejemplos de código

A continuación se muestran dos ejemplos de código para ilustrar cómo se puede usar la librería.

Código 3.1: Análisis del algoritmo *Thompson Sampling* para distintos valores α y β con sorteo de usuarios *Round-robin*. El resultado será una gráfica con la evolución del *recall* acumulado.

```
# Diccionario con los nombres de las columnas del dataset
dict_names = {'userName':'userId','itemName':'trackId','ratingName':'rating'}
# Hiperparámetros
alphas = [1,10,100,1000,1e4]
betas = [1,10,100,1000,1e4]
# Inicialización de la lista de bandits
lst = []
for alpha in alphas:
    for beta in betas:
        bandit = ThompsonSampling(alpha=alpha, beta=beta)
        bandit.read_csv('data/cm100k/ratings.csv', **dict_names)
        lst.append(bandit)
# Creación de un analizador
an = Analysis()
an.execute(lst, epochs=5e5, shuffle='round_robin')
an.recall()
```

Código 3.2: Análisis del coeficiente de Gini en las 100.000 primeras épocas de los algoritmos ε-greedy y UCB con ciertos parámetros. El resultado será una gráfica con la evolución de los coeficientes de Gini.

```
# Diccionario con los nombres de las columnas del dataset
dict_names = {'userName':'userId','itemName':'trackId','ratingName':'rating'}
# Inicialización de bandits
bandits = []
bandits.append(EpsilonGreedy(epsilon=0.2,initial=0))
bandits.append(UCB(c=0.1))
# Introducción de la base de datos
for bandit in bandits:
    bandit.read_csv('data/cm100k/ratings.csv',**dict_names)
# Creación de un analizador
an = Analysis()
an.execute(bandits,1e5,gini_history=True)
an.gini()
```

Si se buscan más ejemplos, en el repositorio de Github se adjuntan *notebooks* con código *IPython* donde se pueden ver todos los experimentos que se han realizado en el Capítulo 5.

3.5. Notación y planteamiento general

Aunque ya se ha expuesto la mayoría de la notación de manera muy extensa en el Capítulo 2, se quiere hacer un resumen sintético de toda la notación genérica que se utilizará a la hora de definir los algoritmos en el siguiente capítulo:

- $\mathcal{A}$ es el conjunto de acciones (brazos) que se pueden elegir en cada unidad de tiempo. En este caso cada acción es la elección de un *item*.
- $\mathcal{U}$ es el conjunto de usuarios en el sistema.
- T es el número de épocas del algoritmo, donde en cada época se recomendará un *item* a un cierto usuario.
- (U_t, A_t) es la dupla que muestra el resultado elegido en cada época t. En concreto, U_t representa el usuario al que recomendamos un *item* y A_t la acción (*item*) seleccionada.
- $R_t \in [0, 1]$ es la recompensa obtenida en tiempo t en relación con la recomendación utilizada.
- $q(a)$ es la recompensa media obtenida por una cierta acción $a \in \mathcal{A}$.
- $Q_t(a)$ es la recompensa media estimada en la época t que se va a obtener por una cierta acción $a \in \mathcal{A}$.

- Es importante recordar la notación de cardinalidad de un conjunto, que es denota habitualmente con el conjunto entre barras. Por ejemplo, el número de usuarios de un sistema de recomendación será $|\mathcal{A}|$

Una vez explicada la notación, el pseudocódigo general que debe seguir todo algoritmo es el del Algoritmo 3.1.

input : Acciones $\mathcal{A}$, usuarios $\mathcal{U}$.
output: Recompensa de cada época
1 **for** $t = 1 \dots T$ **do**
2 $U_t \leftarrow$ recibir_usuario($\mathcal{U}$);
3 $A_t \leftarrow$ recomienda(U_t) $R_t \leftarrow$ obtener_recompensa(A_t);
4 $Q_t(A_t) \leftarrow$ actualiza_valor(A_t,R_t);
5 **end**

Algoritmo 3.1: Procedimento general.

Normalmente, al tratar con algoritmos de recomendación debemos determinar qué se considera una buena o mala recomendación. En esta librería se ha decidido que el límite se encuentra en el punto medio de las valoraciones. Es decir, si llamamos $\mathcal{R}$ al conjunto de valoraciones y $\mathcal{R}^+$ al conjunto de valoraciones positivas:

$$r \in \mathcal{R}^+ \iff r \geq \frac{\max\limits_{a \in \mathcal{R}} a + \min\limits_{a \in \mathcal{R}} a}{2}. \tag{3.1}$$

Asimismo, en esta librería la recompensa que recibe el sistema es binaria: 1 si la recomendación ha sido valorada positivamente y 0 si la recomendación ha sido valorada negativamente o no se ha valorado. Sin embargo, no hay necesidad de seguir estos patrones si no se quiere, ya que también es posible implementar recompensas graduales en función de lo buena que sea una valoración.

4

ALGORITMIA

En este capítulo se mostrará la algoritmia desarrollada en la librería, exponiendo su funcionamiento interno. Todos los algoritmos que se van a clasificar en este capítulo entran en una clasificación binaria: contextuales y no contextuales. Aquellos pseudocódigos que se han considerado demasiado extensos se adjuntan en el Apéndice C.

4.1. Algoritmos no contextuales

Como bien indica la palabra, estos algoritmos no dependen de ningún contexto. Por ello, las recomendaciones que hacen son genéricas para todos los usuarios y son ideales si se busca sencillez a la vez que buenos resultados. Suelen ser los algoritmos clásicos en la literatura de aprendizaje reforzado en bandidos multi-brazo.

4.1.1. ε-greedy

Este algoritmo [Sutton and Barto, 2018, capítulo 2] trata de la manera más sencilla y natural el problema de la explotación-exploración. En primer lugar, se fija un parámetro $\varepsilon \in [0, 1]$ que habitualmente se fija en puntos cercanos al 0. Este parámetro determina la probabilidad de que el algoritmo explore o explote:

- **Exploración**: con una probabilidad ε, el algoritmo escoge una acción aleatoriamente.
- **Explotación**: con una probabilidad $1 - \varepsilon$, el algoritmo escoge la acción codiciosa, es decir, la que tenga mayor recompensa esperada.

La recompensa esperada se calcula como el promedio de las recompensas recibidas. Por lo tanto, para una cierta acción $a \in \mathcal{A}$ en tiempo t donde se ha acertado *hits* veces, se ha fallado *fails* veces y no hemos encontrado datos *misses* veces tenemos que:

$$Q_t(a) = \frac{\sum_{i=0}^{t} R_i \cdot \mathbb{1}_{a_t=a}}{\sum_{i=0}^{t} \mathbb{1}_{a_t=a}} = \frac{hits}{hits + fails + misses}. \tag{4.1}$$

Por lo tanto, en el caso $\varepsilon = 0$ siempre se cogerá la misma acción hasta que se reduzca tanto la recompensa esperada que pase a ser otro brazo la acción codiciosa. En el lado opuesto, si $\varepsilon = 1$ se tiene el algoritmo aleatorio.

Además, se debe fijar un valor $Q_0(a)$ para cada $a \in \mathcal{A}$, es decir, la recompensa estimada a priori cuando no tenemos ninguna información del brazo. Por supuesto, puede ser cualquier valor entre 0 y 1, pero se suelen tomar valores cercanos al 0, ya que en sistemas de recomendación la información a priori suele ser que al usuario no le gusta la película.

Problemas no estacionarios

Para actualizar la recompensa esperada no es necesario disponer de todo el historial de aciertos y fallos del brazo, sino que únicamente necesitamos saber el número de veces que el brazo ha sido elegido. En concreto, en la ecuación (4.2) se observa que, en un momento t en el que se ha escogido el brazo a_t por N-ésima vez:

$$Q_t(a_t) = \frac{\sum_{i=0}^{t} R_i \cdot \mathbb{1}_{a_t=a}}{N} = \frac{\sum_{i=0}^{t-1} R_i \cdot \mathbb{1}_{a_t=a}}{N} + \frac{R_t}{N} = \frac{N-1}{N} Q_{t-1}(a_t) + \frac{R_t}{N}. \tag{4.2}$$

Si un problema es estacionario y queremos que la recompensa esperada converja a un cierto valor, tiene sentido que cada nueva recompensa tenga menos peso en el promedio. En cambio, si el problema es no estacionario y la recompensa esperada va cambiando con el tiempo, deberían tener más peso las nuevas recompensas. Es por esto que en esta sección se muestra una nueva variante del algoritmo capaz de solventar estos problemas.

Para ello, se introduce un nuevo parámetro $\alpha \in [0, 1]$ que refleja el peso que tendrá la última recompensa R_t en el promedio ponderado $Q_t(a_t)$. Por lo tanto, ahora la recompensa esperada se calcula como:

$$Q_t(a_t) = (1 - \alpha) \cdot Q_{t-1}(a_t) + \alpha \cdot R_t. \tag{4.3}$$

Ahora las recompensas más recientes tienen mayor peso, mientras que las más antiguas ponderan casi como peso nulo. En concreto, si se ha elegido el brazo $a \in \mathcal{A}$ N veces tenemos que $Q_0(a)$ tiene

peso $(1-\alpha)^N$, la primera recompensa $\alpha \cdot (1-\alpha)^{N-1}$, la i-ésima recompensa $\alpha \cdot (1-\alpha)^{N-i}$ y la última recompensa α. Es importante ver que los pesos suman 1:

$$(1-\alpha)^N + \sum_{i=0}^{N-1} \alpha \cdot (1-\alpha)^{N-i} = (1-\alpha)^N + \alpha \cdot \frac{1-(1-\alpha)^N}{1-(1-\alpha)} = (1-\alpha)^N + \alpha \cdot \frac{1-(1-\alpha)^N}{\alpha} = 1. \quad (4.4)$$

Aunque ε-greedy es un algoritmo muy sencillo y rápido, tiene un gran problema en lo que a exploración se refiere: explora de forma indiscriminada todas las acciones posibles. Por lo general, existen algoritmos más sofisticados que sacrifican eficiencia en favor de métodos que permitan explorar con más probabilidad aquellos brazos que tengan un mayor potencial.

4.1.2. UCB (Upper Confidence Bound)

Este algoritmo [Sutton and Barto, 2018, capítulo 2] proporciona una versión de ε-greedy mejorada, donde se asume que el problema es estacionario y que tenemos menor necesidad de explorar aquellas acciones donde:

- Ya hemos explorado suficiente, por lo que ya tenemos buena aproximación de la recompensa esperada.
- La recompensa esperada es tan baja que no se espera que mejore aunque tengamos poca información al respecto.

El objetivo es estimar la recompensa de cada brazo y el intervalo de confianza en el que podemos suponer que nuestra estimación es válida. Conforme vaya aumentando el número de veces que hemos elegido un cierto brazo, nuestra estimación será mejor y podremos reducir dicho intervalo de confianza.

En este algoritmo la recompensa esperada $Q_t(a)$ para cada acción $a \in \mathcal{A}$ en tiempo t es calculada como en (4.1), mientras que el número de veces que se ha recomendado dicha acción es $N_t(a)$. Sabiendo esto, la esperanza de la recompensa se puede acotar con una cierta confianza:

$$\mathbf{E}(R_t) = Q_t(a) \pm c\sqrt{\frac{\ln t}{N_t(a)}}, \quad (4.5)$$

donde $c \in [0, \infty]$ es un valor a determinar que indica cómo de grande va a ser el intervalo de confianza. Como se puede observar, el intervalo de confianza se va haciendo más pequeño conforme se escoge más veces la acción a.

Este algoritmo es optimista respecto a los resultados, por lo que siempre se toma como estimación la mayor recompensa del intervalo de confianza (de ahí el nombre del algoritmo). Por ello, la acción elegida en tiempo t es:

$$A_t = \arg\max_{a \in \mathcal{A}} \quad Q_t(a) + c\sqrt{\left(\frac{\ln t}{N_t(a)}\right)}\Bigg(\tag{4.6}$$

Cuando una acción no se ha recomendado nunca $N_t(a)$ es cero. En este caso, se da máxima prioridad a esta acción. Esto significa que en una puesta en frío lo primero que hará el algoritmo es recomendar aleatoriamente todos los *items* disponibles. Como vemos, con este nuevo criterio el algoritmo es capaz de manejar con cierta soltura la incertidumbre que existe con la recompensa esperada calculada.

4.1.3. CNAME

Algoritmos como ε-greedy no aprovechan como deberían todo el feedback que se les da, mientras que los algoritmos como UCB dejan de funcionar tan bien en problemas a gran escala. El algoritmo CNAME [Zhang et al., 2018] fue diseñado para conciliar los dos puntos de vista, de forma que se utilicen los resultados obtenidos para estimar correctamente recompensas y para ajustar el coeficiente de exploración.

En el Algoritmo C.1, el parámetro calculado p es el coeficiente de exploración, que es mayor conforme mayor sea el parámetro inicial w y conforme menor sea el valor m (número de épocas que se ha accionado el brazo con menor recompensa). De esta forma, la probabilidad de exploración se va a adaptando, siendo mayor al principio y reduciéndose con el tiempo.

De esta forma, se utiliza la visión de ε-greedy, donde se recomiendan elementos aleatorios a veces con el fin de facilitar la exploración de los distintos brazos. Sin embargo, la filosofía de UCB se puede entrever con la reducción de la probabilidad de exploración, ya que tras un gran número de épocas tenemos una gran confianza en nuestros resultados.

4.1.4. Gradient Bandit

Este algoritmo [Sutton and Barto, 2018, capítulo 2] aprovecha la clásica idea en aprendizaje automático del descenso gradiente para crear preferencias hacia el accionamiento de cada brazo. Se puede utilizar tanto para problemas estacionarios como para no estacionarios, aunque requiere unas pequeñas variaciones.

En este algoritmo, en tiempo t cada acción $a \in \mathcal{A}$ tiene una preferencia $H_t(a)$. Con esta idea buscamos distanciarnos del concepto *recompensa esperada* utilizado anteriormente, ya que ahora solo buscamos un valor numérico relativo que dé prioridad a aquellos brazos que muestren mejor comportamiento. Inicialmente, $H_t(a) = 0$ para toda acción, por lo que todas tienen la misma probabilidad de ser elegidas. La probabilidad de elección de cada brazo se determina mediante la **distribución soft-max**:

$$\mathbf{P}(A_t = a) = \frac{e^{H_t(a)}}{\sum_{b \in \mathcal{A}} e^{H_t(b)}}, \tag{4.7}$$

donde claramente la suma de todas las probabilidades suma 1.

Una vez elegido el brazo A_t que va a recomendar un cierto *item* toca actualizar las preferencias H_t según la recompensa R_t obtenida. Para calcularla, necesitamos dos ingredientes:

- Una constante de aprendizaje $\alpha > 0$, que tiene la misma función que en el descenso gradiente tradicional.
- La recompensa media $\overline{R}_t$. En caso de que estemos en un problema estacionario, se debe utilizar la fórmula (4.2), mientras que si no lo es debemos utilizar (4.3) (fijando su correspondiente parámetro de pesos llamado *avgRate*).

Una vez determinados estos valores, las nuevas preferencias se calculan como siguen:

$$H_{t+1}(a) = \begin{cases} H_t(a) + \alpha(R_t - \overline{R}_t)(1 - \mathbb{P}(A_t = a)) & \text{si } a = A_t \\ H_t(a) - \alpha(R_t - \overline{R}_t)\mathbb{P}(A_t = a) & \text{en cualquier otro caso.} \end{cases} \tag{4.8}$$

Como vemos, la preferencia del brazo seleccionado se actualiza en función de cómo sea la recompensa respecto a la media de recompensas, mientras que el resto de preferencias se mueven en dirección contraria. En concreto, el "movimiento total" de las preferencias es 0.

4.1.5. Thompson Sampling

Thompson Sampling [Chapelle and Li, 2011] es un método muy general en aprendizaje reforzado para problemas de valores estacionarios. Este método asume que la recompensa depende de la acción escogida $a \in \mathcal{A}$, un cierto contexto x y un conjunto de parámetros θ_t que solo somos capaces de estimar, pero de los que conocemos su distribución condicionada a los resultados $\{(A_i, X_i, R_i\}_{i=0}^{t}$ conocidos.

En cada iteración a tiempo t con un cierto contexto x_t se escoge aleatoriamente un valor para θ_t de acuerdo con la distribución mencionada. Una vez conocida, para cada acción $a \in \mathcal{A}$ se puede calcular la recompensa esperada teniendo en cuenta todos los datos de los que disponemos. Se adjunta el algoritmo 4.1 que generaliza toda versión de *Thompson Sampling*.

Ahora, debemos generalizar el método a bandidos multi-brazo en sistemas de recomendación. En concreto, nuestro contexto es el usuario escogido y el parámetro θ_t es la recompensa estimada de cada acción. Además, nuestras recompensas solo adoptan dos valores: $R_t \in \{0, 1\}$, por lo que su distribución es una Bernouilli cuya esperanza hemos estimado mediante θ_t.

input : Acciones $\mathcal{A}$, usuarios $\mathcal{U}$.
output: Recompensa de cada época

D = {};
for $t = 1 \dots T$ **do**
  $U_t \leftarrow$ recibir_usuario($\mathcal{U}$);
  $\mathcal{A}^* \leftarrow$ acciones_no_exploradas(U_t);
  $X_t \leftarrow$ recibir_contexto(t);
  $\theta_t \leftarrow$ random_distr($\mathbb{P}(\theta|D)$);
  $A_t \leftarrow \arg\max_{a \in \mathcal{A}^*} \mathbf{E}(R_t|X_t, a, \theta_t)$;
  $R_t \leftarrow$ obtener_recompensa(A_t);
  $D \leftarrow D \cup \{(X_t, A_t, R_t)\}$;
end

Algoritmo 4.1: Thompson Sampling.

Queda especificar $\mathbb{P}(\theta|D)$. Para ello, pongamos que la i-ésima componente de θ_t es la recompensa estimada de una cierta acción $a \in \mathcal{A}$. Entonces, su distribución se puede modelizar como una *Beta*:

$$(\theta_t)_i \sim Beta(\alpha + hits, \beta + fails), \tag{4.9}$$

donde $\alpha, \beta \in (0, \infty)$ son los valores de la distribución a priori y $hits, fails \in \mathbb{N}$ son los aciertos y fallos contabilizados para la acción escogida. En realidad, α y β y son más que el tipo de inicialización que queramos poner: optimista o pesimista. Si $\alpha \gg \beta$, es muy probable obtener valores cercanos a 1, suponiendo así resultados muy optimistas. Por el contrario, si $\beta \gg \alpha$ es muy probable obtener valores cercanos a 0, suponiendo así resultados muy pesimistas. Con estos ingredientes, el algoritmo *Thompson Sampling* queda especificado en el Algoritmo 4.2.

La distribución beta funciona correctamente gracias a su naturaleza. Si disponemos de una distribución $Beta(a, b)$, su media resulta ser $\frac{a}{a+b}$ (perfecto para aproximar la recompensa esperada) y su varianza es $\frac{ab}{(a+b)^2(a+b+1)}$, que va decreciendo según a y b incrementan. En concreto, en sistemas de recomendación la probabilidad a priori de que un *item* dé una recompensa positiva es baja, por lo que se recomienda inicializar α con un valor pequeño (como 1) y β con un valor elevado (como 100 o 1000).

4.1.6. EXP3

Otro tipo de bandidos multi-brazo son los **bandidos adversario**. Estos algoritmos se basan en simular un juego donde tenemos varias acciones de recompensas desconocidas y queremos maximizar nuestro éxito conforme realicemos experimentos. Hasta ahora, los algoritmos se basaban en estimar estadísticamente las recompensas y elegir la mayor estimación suponiendo un cierto *optimismo* sobre su incertidumbre. En cambio, con este modelo se asume que nuestro modelo depende de cómo vaya progresando, por lo que se puede estar eligiendo el algoritmo con peor recompensa estimada, así que estos algoritmos presupone un cierto *pesimismo* en cuanto a estas recompensas.

input : Acciones $\mathcal{A}$, usuarios $\mathcal{U}$, valores iniciales α, β.
output: Recompensa de cada época

$N = |\mathcal{A}|$;
$H = \mathbf{0}^N$; /* array de aciertos de cada brazo */
$F = \mathbf{0}^N$; /* array de fallos de cada brazo */
for $t = 1 \dots T$ **do**
  $U_t \leftarrow$ recibir_usuario($\mathcal{U}$);
  $\mathcal{A}^* \leftarrow$ acciones_no_exploradas(U_t);
  for $a \in \mathcal{A}^*$ **do**
    $\theta_a \leftarrow$ random_distr($Beta(\alpha + H_a, \beta + F_a)$);
  end
  $A_t \leftarrow \arg\max_{a \in \mathcal{A}^*} \theta_a$;
  $R_t \leftarrow$ obtener_recompensa(A_t);
  if $R_t = 1$ **then** $H_{A_t} = H_{A_t} + 1$;
  else $F_{A_t} = F_{A_t} + 1$;
  ;
end

Algoritmo 4.2: Thompson Sampling con distribución Beta.

El algoritmo *EXP3* [Seldin et al., 2013] pretende estimar la probabilidad de que se consiga una buena recompensa escogiendo una cierta acción. Conforme a estas probabilidades, se hace una elección aleatoria ponderada de la acción que vaya a ser escogida. Tras conocer su recompensa, se recalcula la probabilidad de cada brazo en función del resultado obtenido.

input : Acciones $\mathcal{A}$, usuarios $\mathcal{U}$, valor inicial $\alpha \in [0, 1]$.
output: Recompensa de cada época

Se inicializa el vector w de pesos, donde $w_i = 1 \ \forall i \in \mathcal{A}$;
for $t = 1 \dots T$ **do**
  $U_t \leftarrow$ recibir_usuario($\mathcal{U}$);
  $\mathcal{A}^* \leftarrow$ acciones_no_exploradas(U_t);
  for $a \in \mathcal{A}^*$ **do**
    $p_a \leftarrow (1 - \alpha)\frac{w_a}{\sum_{i \in \mathcal{A}^*} w_i} + \alpha \frac{1}{|\mathcal{A}^*|}$;
  end
  $A_t \leftarrow$ Elección aleatoria en $\mathcal{A}^*$ respecto a la distribución de los p;
  $R_t \leftarrow$ obtener_recompensa(A_t);
  $w_a \leftarrow w_a e^{\frac{\alpha R_t}{|\mathcal{A}^*|}}$;
end

Algoritmo 4.3: Algoritmo EXP3.

Se puede observar en detalle el pseudocódigo en el Algoritmo 4.3. El parámetro a fijar α es la constante de exploración y por tanto conviene que se vaya rebajando conforme avanzan las épocas. En la librería implementada se ha optado por la siguiente expresión:

$$\alpha(t) = e^{-\frac{t}{k \cdot |\mathcal{A}^*|}}, \tag{4.10}$$

donde k es el parámetro a determinar. Este valor α es próximo a 1 para valores pequeños de t y va reduciéndose exponencialmente según avanzan las épocas (aunque tarda k veces el número de brazos en alcanzar $1/e$).

Conviene darse cuenta de que las probabilidades de los brazos solo cambian si la valoración a sido positiva. Por otro lado, si k es un valor grande, se prioriza la exploración durante muchas más épocas, aunque a la vez se da mucha más importancia a las valoraciones positivas (línea 10 del Algoritmo 4.3). En cambio, para k pequeños se deja rápido la fase de exploración a la vez que no se da tanta importancia a las valoraciones positivas.

4.2. Algoritmos contextuales

Hasta ahora, todos los algoritmos se basaban en tratar todos los *items* como elementos incorrelados que podían ser elegidos. Sin embargo, esto no es así, ya que *items* o circunstancias similares dan lugar a decisiones parecidas.

En los algoritmos que siguen se utiliza un contexto intrínseco al *item* para mostrar semejanzas y diferencias entre ellos, por lo que no evoluciona con el tiempo. En concreto, se recomienda que no tenga una gran dimensión, ya que se van a fabricar matrices de dicha dimensión y se tendrán que invertir, por lo que puede resultar muy costoso tener matrices demasiado grandes. Debido a su gran extensión, los pseudocódigos de estos algoritmos se mostrarán en el Apéndice C.

4.2.1. LinUCB

El algoritmo **LinUCB** [Li et al., 2010] es una versión del algoritmo UCB donde las recompensas esperadas se estiman condicionándolas a la información que nos da el contexto. Para hallar la recompensa estimada a través del contexto, se utilizan los métodos que se detallan en el Apéndice B. Dicha estimación permite obtener también un intervalo de confianza, cuya amplitud viene determinada por un parámetro α a fijar.

Como ocurre en todos los algoritmos UCB, se escoge el resultado más optimista dentro del intervalo de la recompensa esperada, por lo que el parámetro α puede ser entendido como una constante de exploración. De esta forma, el pseudocódigo queda descrito en el Algoritmo C.2.

Conviene darse cuenta de que este algoritmo (al igual que los siguientes) tiene vectores de gran dimensión (sobre 20) y matrices cuadradas de la misma dimensión. El algoritmo obliga a multiplicar e invertir estas matrices en cada época, por lo que el coste computacional de estos algoritmos es mucho mayor que el de los anteriores.

4.2.2. DynUCB

Aunque el algoritmo LinUCB incluía una mejora a la hora de asociar *items* similares, incluía una simplificación con la que se han tratado a todos los algoritmos hasta ahora: a todos los usuarios se les recomienda lo mismo, sin ningún tipo de personalización. El algoritmo DynUCB [Nguyen and Lauw,] pretende cambiar esto mediante ***clustering* de usuarios**.

En este algoritmo, se debe fijar el número de *clusters* K al comenzar. Además, a cada usuario $u \in \mathcal{U}$ se le destina una matriz M_u y un vector b_u equivalentes a los del problema LinUCB. Al comienzo, cada usuario es asignado a un *cluster* aleatorio, donde cada uno tiene una matriz M_k y un vector b_k definidos como:

$$M_k = \mathbf{I}_{d\times d} + \sum_{u\in C_k}(M_u - \mathbf{I}_{d\times d}) \tag{4.11a}$$

$$b_k = \sum_{u\in C_k}(b_u, \tag{4.11b}$$

donde d es la dimensión del contexto. Con estos resultados, la distancia entre un cierto *cluster* C_k y un cierto usuario u es:

$$d(u, C_k) = ||M_u^{-1}b_u - M_k^{-1}b_k||, \tag{4.12}$$

y tratará de minimizarse cada vez que se recomiende un *item* a un usuario. Por tanto, tras actualizar un cierto M_u y b_u se comprobará si ahora el *cluster* más próximo al usuario es en el que está o es otro. Si es otro *cluster*, se deben recalcular los valores de M_k y b_k del nuevo *cluster* y del antiguo *cluster* de acuerdo con (4.11a) y (4.11b). Todo este proceso se formaliza en el Algoritmo C.3.

4.2.3. CLUB

El algoritmo anterior mantiene un número estático de *clusters*. Sin embargo, el algoritmo CLUB [Gentile et al., 2014] (*Cluster of Bandits*) trata este problema, aumentando el número de clusters conforme se detectan suficientes diferencias contextuales entre los usuarios.

Al igual que en el algoritmo anterior, existe una matriz M_u y un vector b_u asociados a cada usuario, definidas igual que en el algoritmo *LinUCB*, donde todos los usuarios comienzan formando parte de un mismo *cluster*. Dicho cluster también tiene una matriz M_k y un vector b_k asociados calculados como en (4.11a) y en (4.11b).

Pasemos ahora a ver cómo se realiza la segmentación en varios *clusters*. El conjunto de usuarios pasa a interpretarse como un grafo no dirigido donde los nodos son dichos usuarios y las aristas una conexión que muestra cierta relación. El grafo comienza siendo completo (todos los nodos están conectados con todos los nodos) y se van eliminando aquellas aristas en las que se considera que no

hay suficiente relación entre los usuarios.

Siendo el vector de predicción de un cierto usuario $M_u^{-1}b_u$, denominemos a su margen de confianza CB_u [1]. Una buena aproximación de dicho valor es:

$$CB_u = \beta\sqrt{\frac{1+\log(1+T_u)}{1+T_u}}, \tag{4.13}$$

donde $\beta > 0$ es un parámetro de eliminación de aristas fijado y T_u el número de épocas que se ha escogido un *item* para el usuario $u \in \mathcal{U}$.

Sabiendo esto, el criterio que se va a utilizar para saber si se debe eliminar una arista entre dos nodos u_1, u_2 es:

$$||M_{u_1}^{-1}b_{u_1} - M_{u_2}^{-1}b_{u_2}|| > CB_{u_1} + CB_{u_2}. \tag{4.14}$$

Es decir se considera que dos usuarios no están relacionados si la norma de la diferencia entre los vectores de predicción es mayor que la suma de sus valores de confianza. **Se forman nuevos *clusters* según se obtengan nuevas componentes conexas**. Como notación, K_t será el número de *clusters* en tiempo t y $\mathbf{C}_t = \{C_1 \dots C_{K_t}\}$ dichos *clusters*. Con estos ingredientes y lo ya visto en el algoritmo de DynUCB C.3 se forma el nuevo Algoritmo C.4.

Existe una pequeña medida que aumenta la eficiencia del algoritmo: no es necesario recomputar todas las componentes conexas en la línea 28. Solo es necesario que, cada vez que se elimine una arista entre dos usuarios, se compruebe si sigue existiendo un camino entre dichos usuarios (mediante búsqueda en profundidad, por ejemplo). Si sigue habiendo un camino hay el mismo número de componentes conexas; si no lo hay se ha creado una nueva componente conexa.

[1] Del inglés *Confidence Bound*

5

Experimentos y resultados

En este apartado se van a analizar los algoritmos descritos en el apartado anterior. Para ello, las librerías utilizadas serán:

- *MovieLens 100K* [Harper and Konstan, 2015]. Una de las bases de datos más conocidas en el mundo de los sistemas de recomendación. En ella, se incluyen 100.836 valoraciones del 0 al 5 de 671 usuarios a 9.724 películas. Se consideran positivos aquellos *ratings* que sean 2,5 o superior. Para guardar el contexto se utilizan los géneros de cada película, ya que tienen una dimensionalidad reducida.

- *CM100K* [Cañamares and Castells, 2018]. Una base de datos en las que se recomiendan canciones a los usuarios sin sesgo (es decir, no hay un recomendador detrás que pueda dar más prioridad a ciertos *items*). Contiene 103.584 valoraciones de 1.084 canciones y 1.054 usuarios.

Para dar fluidez a la lectura y mostrar resultados unificados, se presentarán los resultados con la base de datos *MovieLens 100K*, mientras que los experimentos con *CM100K* se dejan indicados en el Apéndice D, dado que solo se pueden hacer *bandits* no contextuales con él y los resultados son más planos.

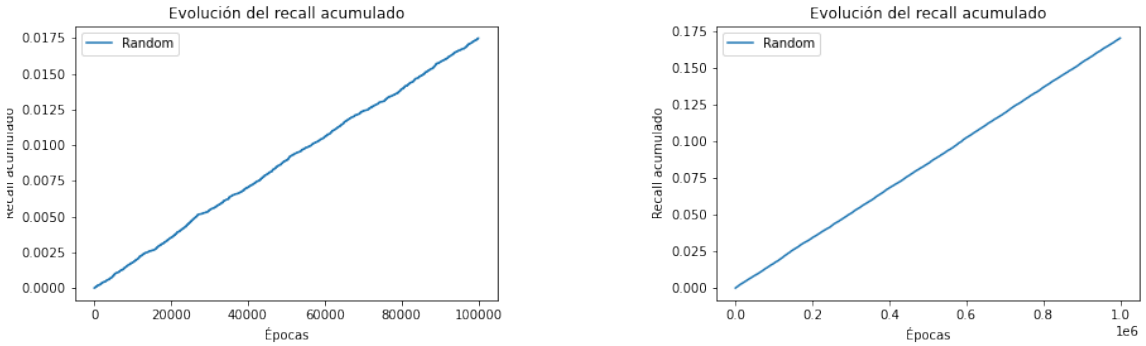
Recordemos también que todos los problemas tratados son **estacionarios**, ya que no nos llega información nueva en ningún momento, sino que partimos de toda la base de datos y el algoritmo debe ir hallando recomendaciones positivas.

5.1. Análisis del *recall* acumulado

Esta sección se va a dividir en dos partes: primero, se va a hacer un análisis de hiperparámetros de la base de datos *movieLens*; después, se va a comparar cada algoritmo con sus mejores hiperparámetros.

5.1.1. Búsqueda en rejilla de hiperparámetros

Para marcar una referencia a todos los algoritmos que vienen, se adjunta cómo clasifica el recomendador aleatorio en la Figura 5.1. Se espera que todos los algoritmos puedan funcionar mejor o peor, pero que siempre superen de sobra la barrera de la predicción aleatoria.



(a) Con 100.000 épocas.

(b) Con un millón de épocas

Figura 5.1: Recall acumulado del clasificador aleatorio en la base de datos *MovieLens*.

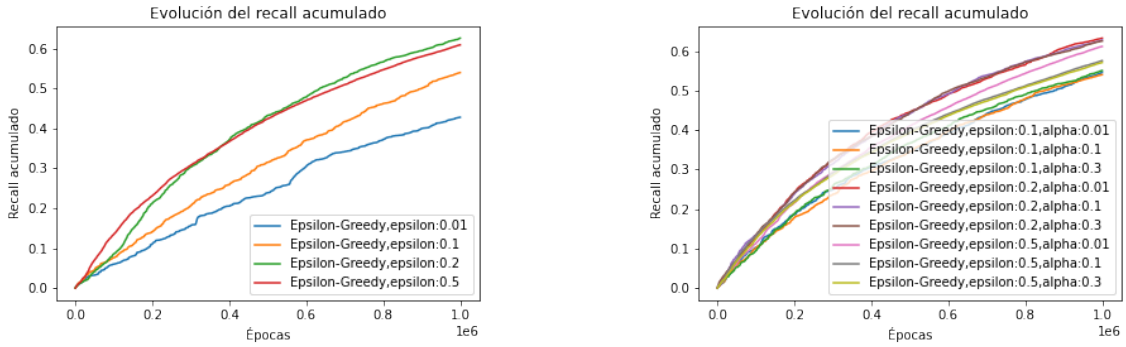
Como vemos, es capaz de predecir un $1{,}75\,\%$ de las valoraciones postivas cada 100.000 épocas, por lo que al alcanzar el millón ha predicho un $17{,}5\,\%$. Además, se puede ver que la gráfica es lineal, por lo que si fuera necesario podríamos interpolar para obtener el *recall* acumulado tras un cierto número de épocas.

Algoritmos no contextuales

Comencemos con el algoritmo ε**-greedy**. Como ya se ha dicho, este es un problema estacionario, pero para verlo visualmente daremos más peso a las nuevas predicciones, como en un problema no estacionario. Lo esperado es que el *recall acumulado* no mejore o incluso disminuya.

En la Figura 5.2 se puede observar este fenómeno: prácticamente parece irrelevante el valor α que se escoja, ya que solo el valor ε influye en la evolución del *recall* acumulado. Además, parece que un ε alto influye positivamente en el *recall* (como $\varepsilon = 0{,}2$), aunque uno demasiado alto (como $\varepsilon = 0{,}5$) empieza a perjudicar al algoritmo tras muchas épocas. Esto tiene sentido, ya que una vez que se ha explorado suficiente ha llegado el momento de explotar.

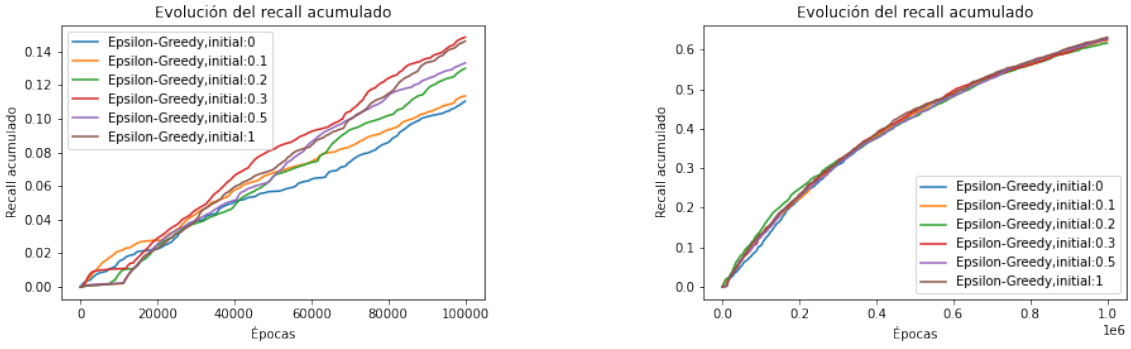
Además, este algoritmo permite tener inicializaciones optimistas o pesimistas, tal y como se muestra en la Figura 5.3. Como es normal, estas inicializaciones solo afectan a una puesta en frío, por lo que solo puede notarse diferencia tras un número pequeño de épocas. Dentro de esto, parece que dan mejor resultado las inicializaciones optimistas, permitiendo así que los *items* no explorados sean elegidos con más facilidad.



(a) Suponiendo que es un problema estacionario.

(b) Suponiendo que es un problema no estacionario.

Figura 5.2: Recall acumulado de ε-greedy tras un millón de épocas en la base de datos *MovieLens*.

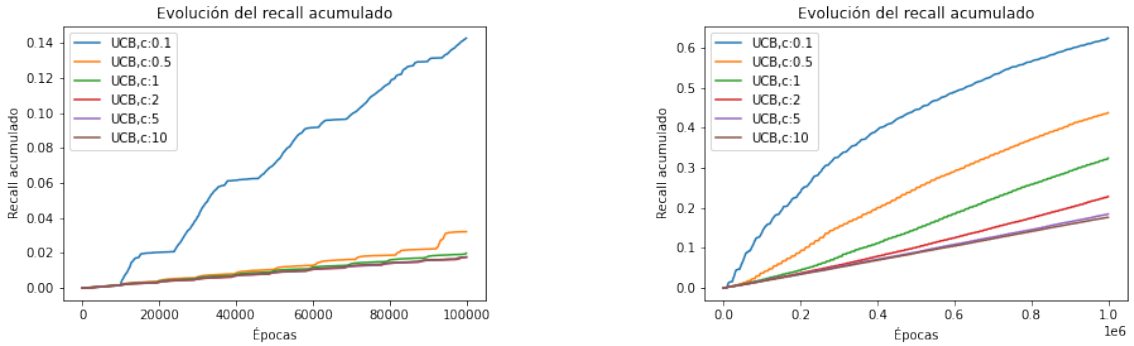


(a) Con 100.000 épocas.

(b) Con un millón de épocas.

Figura 5.3: *Recall* acumulado de ε-greedy para distintos valores iniciales en la base de datos *MovieLens*.

Ahora pasemos al algoritmo **UCB**. Una vez visto que estamos en un problema estacionario, solo nos queda fijar el parámetro $c > 0$, que fija el intervalo de confianza que se da de margen a la recompensa esperada. En la Figura 5.4 se muestran estos resultados.



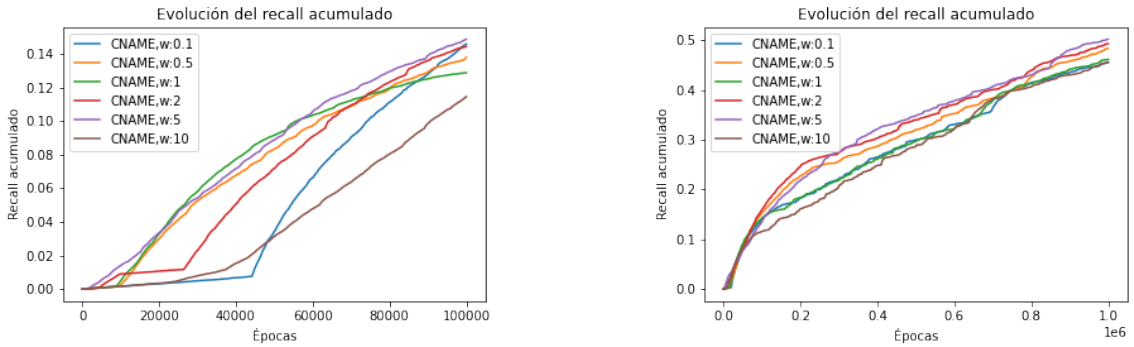
(a) Con 100.000 épocas.

(b) Con un millón de épocas.

Figura 5.4: *Recall* acumulado de UCB en la base de datos *MovieLens*.

Como vemos, el valor c debe ser muy bajo. De hecho, cuando alcanza valores altos (a partir de 2) no se diferencia del recomendador aleatorio.

Veamos ahora el algoritmo **CNAME**, que mezcla los conceptos de ε-greedy y UCB. En él, el solo hay que fijar el parámetro de exploración $w > 0$, cuyos resultados se ven en la Figura 5.5.



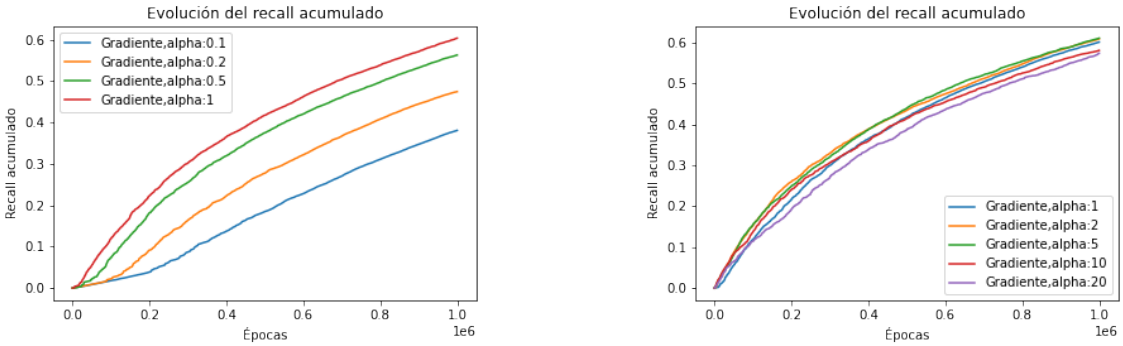
(a) Con 100.000 épocas.

(b) Con un millón de épocas.

Figura 5.5: *Recall* acumulado de CNAME en la base de datos *MovieLens*.

Como vemos, el mejor valor en este caso es intermedio: un valor que oscile entre el 2 y el 5 parece una buena opción. Con más o menos valor parece que pierde *recall*, pero una mala elección del parámetro no desestabiliza tanto los buenos resultados como en el caso de ε-greedy y UCB.

Ahora pasemos al algoritmo **gradiente**. De nuevo, como estamos en un problema estacionario podemos reducir el problema a un único hiperparámetro: la tasa de aprendizaje $\alpha > 0$. Veamos su comportamiento con distintos valores en la Figura 5.6.



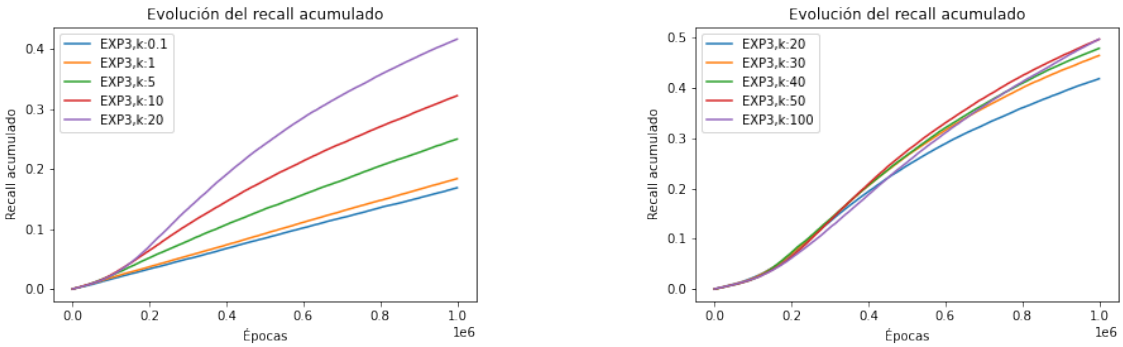
(a) Para valores α pequeños.

(b) Para valores α grandes.

Figura 5.6: *Recall* acumulado del algoritmo Gradiente en la base de datos *MovieLens*.

Como vemos, los valores altos son mejores que los valores bajos. En el caso de que se haya escogido una constante demasiado pequeña, nuestro algoritmo no aprenderá casi nada y será muy parecido al caso aleatorio. En cambio, al igual que en CNAME una tasa de aprendizaje demasiado alta no parece perjudicar mucho al *recall*. Dicho esto, la mejor elección está en un valor intermedio entre 2 y 5.

Ahora, veamos el algortimo **EXP3**. En él, el único valor que hay que analizar es el parámetro que define cómo de lento decrece la tasa de aprendizaje (recordemos que decrece de forma exponencial). Sabiendo esto, se muestra cómo evoluciona el algoritmo con distintos valores en la Figura 5.7



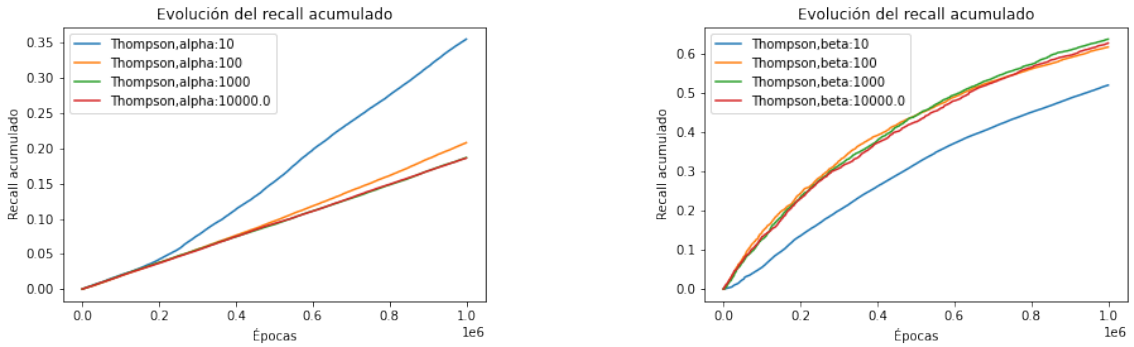
(a) Para valores pequeños.

(b) Para valores grandes.

Figura 5.7: *Recall* acumulado de EXP3 en la base de datos *MovieLens*.

Como vemos, los valores más altos son los que dan mejores resultados (mínimo $k = 30$). En concreto, $k = 0{,}1$ se comporta exactamente igual que el algoritmo aleatorio. Todo esto parece indicar que es mejor que la tasa de aprendizaje decrezca muy lentamente y dé mucha importancia a las valoraciones positivas. Con todo lo dicho, se va a tomar como mejor valor $k = 50$.

Finalmente, veremos el algoritmo ***Thompson Sampling***. En él, existen los parámetros α y β a determinar, pero ambos están relacionados con suponer una inicialización optimista o pesimista. En concreto, la mayoría de datos no están definidos, por lo que en general se predice un fallo. Por ello, es de esperar que el algoritmo muestre buenos resultados si empieza con una inicialización pesimista, como la que se muestra en la Figura 5.8.



(a) Inicialización optimista ($\beta = 1$).

(b) Inicialización pesimista ($\alpha = 1$).

Figura 5.8: *Recall* acumulado de *Thompson Sampling* en la base de datos *MovieLens*.

Como puede verse, los mejores resultados se muestran con una inicialización pesimista, en concreto con un $\alpha = 1$ y un $\beta \geq 100$. A partir de ahí, cuanto más optimista sea una inicialización peores *recall* acumulados obtenemos, llegando incluso al recomendador aleatorio cuando $\alpha \geq 1000$ y $\beta = 1$.

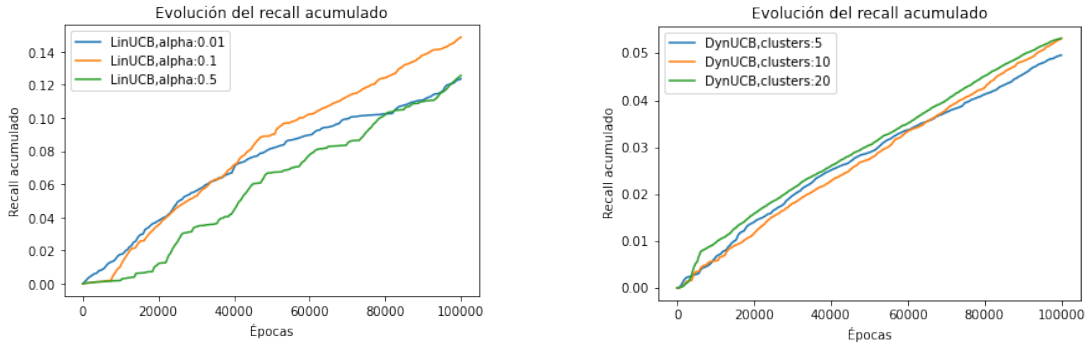
Algoritmos contextuales

Este tipo de algoritmos tienen ejecuciones mucho más costosas (hasta 3 horas cada ejecución con 100.000 épocas). Por ello, se ha reducido el número de épocas a 100.000 y se va a ser más estricto a la hora de seleccionar hiperparámetros.

Comencemos con el algoritmo **LinUCB**, que solo tiene un parámetro: la constante de confianza $\alpha > 0$. Como ya se ha visto en el algoritmo UCB, las mejores constantes de confianza son pequeñas. Como siguen el mismo principio, se van a tomar valores pequeños también en este algoritmo, tal y como se muestra en la Figura 5.9(a). En este caso, la mejor opción es el valor intermedio de los tres elegidos: $\alpha = 0{,}1$.

Veamos ahora el algoritmo **DynUCB**, que tiene dos hiperparámetros: la constante de confianza $\alpha > 0$ y el número de *clusters*. Sin embargo, como α se rige por el mismo principio que en el algoritmo LinUCB se va a asumir que el mejor valor es $0{,}1$. Con esto, el ajuste de hiperparámetros solo afecta al número de *clusters*, como se muestra en la Figura 5.9(b). En este caso parece que no afecta mucho el número de *clusters* utilizados, pero un número grande (como 10 o 20) es una mejor opción.

Veamos por último el algoritmo **CLUB**, que de nuevo tiene dos hiperparámetros: la constante de



(a) Inicialización optimista ($\beta = 1$).

(b) Inicialización pesimista ($\alpha = 1$).

Figura 5.9: *Recall* acumulado de *LinUCB* y *DynUCB* en la base de datos *MovieLens*.

confianza $\alpha > 0$ y la constante de destrucción de nodos $\beta > 0$. Como α ya es conocido por seguir el mismo principio, se mantiene en $0{,}1$, mientras que β va a ser el hiperparámetro a analizar. En la Figura 5.10 se observa este análisis, donde se puede ver que el mejor valor de β es 1.

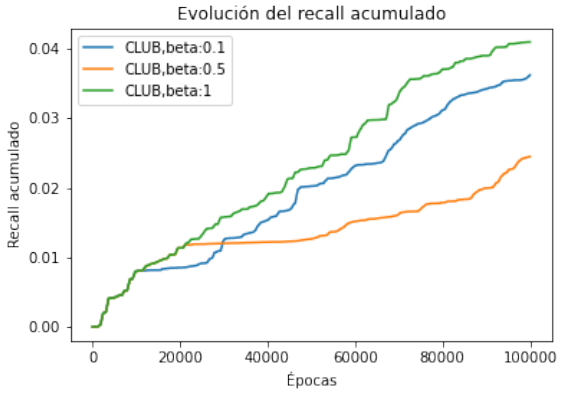


Figura 5.10: *Recall* acumulado de CLUB en la base de datos *MovieLens*.

5.1.2. Comparativa de algoritmos

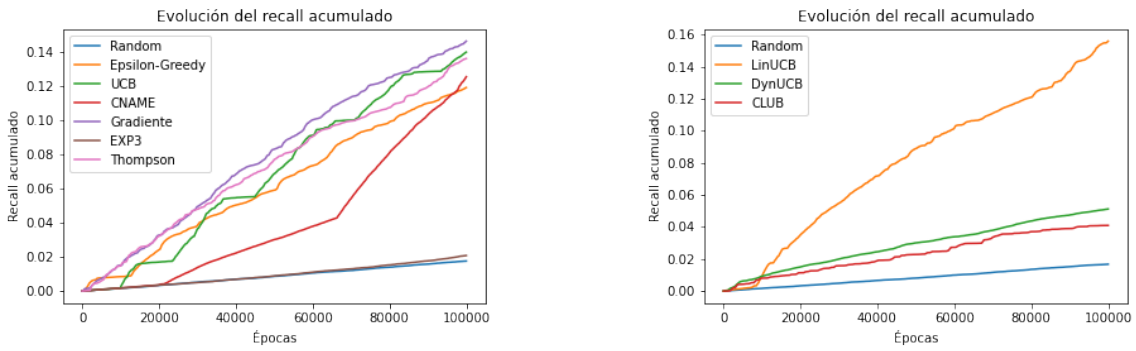
Dados los algoritmos anteriores, los hiperparámetros elegidos de cada algoritmo son los que se muestran el la Tabla 5.1.

Sabiendo estos datos, se va a comparar la evolución de los *recall* acumulados de cada uno de los algoritmos. Haremos dos análisis: por un lado, ejecutaremos 100.000 épocas en todos los algoritmos para establecer una comparación con la Figura 5.11 y la Tabla 5.2. Después, estableceremos comparativa entre los algoritmos contextuales con ejecuciones de un millón de épocas en la Figura 5.12 y en la Tabla 5.3.

En los algoritmos contextuales se puede ver que tanto DynUCB como CLUB muestran datos mu-

Algoritmo	Selección
ε-greedy	$\varepsilon = 0{,}2$; $initial = 0{,}3$
UCB	$c = 0{,}1$
CNAME	$w = 5$
Gradiente	$\alpha = 5$
EXP3	$k = 50$
Thompson Sampling	$\alpha = 1$; $\beta = 1000$
LinUCB	$\alpha = 0{,}1$
DynUCB	$\alpha = 0{,}1$; $clusters = 10$
CLUB	$\alpha = 0{,}1$; $\beta = 1$

Tabla 5.1: Tabla de selección de hiperparámetros



(a) Algoritmos no contextuales.

(b) Algoritmos contextuales.

Figura 5.11: Comparativa del *recall* acumulado en los algoritmos optimizados en *MovieLens*.

Rank	Algoritmo	*Recall* ac.
1	Gradiente	15,1 %
2	CNAME	14,6 %
3	UCB	13,9 %
4	ε-greedy	13,0 %
5	Thompson	12,2 %
6	EXP3	2,1 %

(a) Algoritmos no contextuales.

Rank	Algoritmo	*Recall* ac.
1	LinUCB	15,6 %
2	DynUCB	4,9 %
3	CLUB	4,1 %

(b) Algoritmos contextuales.

Tabla 5.2: Ranking de los algoritmos según el *recall* acumulado en la base de datos *MovieLens* tras 100.000 épocas.

cho peores que el promedio de los algoritmos no contextuales. Hay varias hipótesis que justifican este comportamiento: puede que se parezca demasiado al recomendador aleatorio o que solo esté recomendando unos pocos *items*. Las soluciones a este problema abren una línea de investigación (se sugiere probar con contextos más sofisticados o probar con bases de datos nuevas). Sin embargo, LinUCB no solo funciona bien, sino que es el mejor recomendador de todos con esta métrica.

Respecto a los algoritmos no contextuales, podemos ver que el algoritmo Gradiente es el mejor de todos, mientras que EXP3 es el peor (casi igual que el aleatorio). Sin embargo, ya sabemos por su análisis de hiperparámetros en la Figura 5.7 que este algoritmo tarda mucho en arrancar, por lo que no podemos juzgarlo como el resto con solo 100.000 épocas. El resto de algoritmos parecen funcionar bastante bien, aunque este comportamiento puede ser errático con tan pocas épocas. Aunque este análisis no puede llevarse a cabo en los algoritmos contextuales, se adjunta el análisis de un millón de épocas para algoritmos no contextuales en la Figura 5.12 y en la Tabla 5.3.

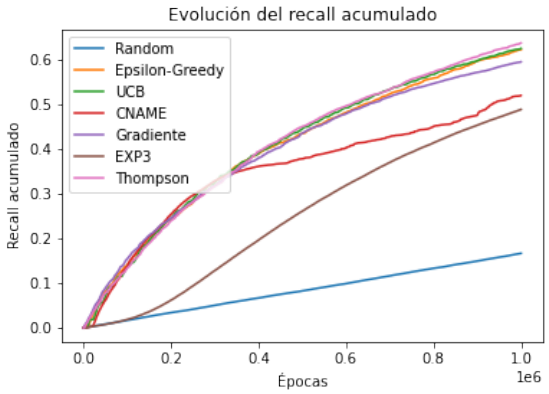


Figura 5.12: *Recall* acumulado de algoritmos en la base de datos *MovieLens* tras un millón de épocas.

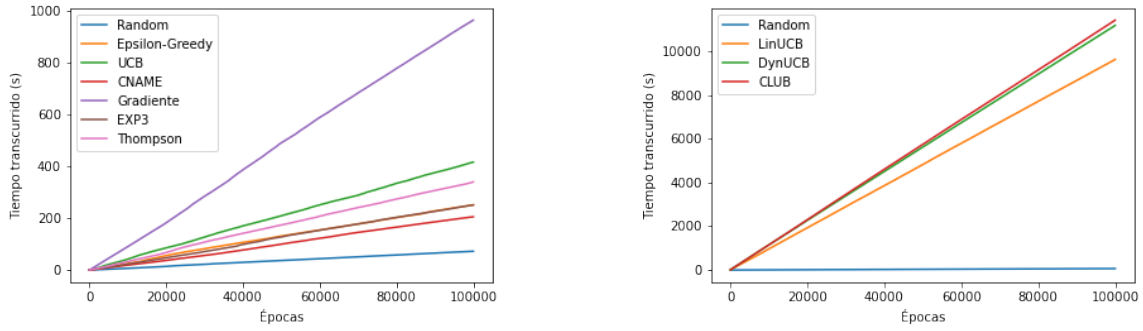
Rank	Algoritmo	*Recall* ac.
1	Thompson	$63{,}8\,\%$
2	UCB	$62{,}6\,\%$
3	ε-greedy	$62{,}3\,\%$
4	Gradiente	$59{,}6\,\%$
5	CNAME	$52{,}0\,\%$
6	EXP3	$48{,}9\,\%$

Tabla 5.3: Ranking de algoritmos no contextuales en *MovieLens* para un millón de épocas.

Con tantas épocas, han cambiado un poco las cosas: ahora casi todos los algoritmos dan los mismos resultados, pero CNAME ha empezado a no recomendar muy bien tras las 300.000 primeras épocas. Esto nos indica que CNAME actúa bien solo para problemas a corto plazo, mientras que el resto son polivalentes. Por otro lado, EXP3 sigue siendo el peor, pero ha mejorado mucho sus resultados y es posible que llegue a superar al resto a largo plazo.

5.2. Análisis del tiempo de ejecución

Un aspecto muy interesante de estos algoritmos es su tiempo de ejecución, ya que una pequeña mejora del *recall* acumulado en un algoritmo puede no ser de gran ayuda si conlleva un gran aumento en el tiempo de entrenamiento. Por ello, se muestran a continuación gráficos comparativos que permitan ver cuáles son los algoritmos más sencillos a nivel computacional, además de ver que los tiempos de ejecución son lineales con las épocas (un apunte muy importante).



(a) Algoritmos no contextuales.

(b) Algoritmos contextuales.

Figura 5.13: Comparativa del tiempo de ejecución tras 100.000 épocas en los algoritmos optimizados en *MovieLens*.

Rank	Algoritmo	Tiempo
1	CNAME	0:03:31
2	ε-greedy	0:03:48
3	EXP3	0:04:15
4	Thompson	0:05:49
5	UCB	0:06:41
6	Gradiente	0:16:00

(a) Algoritmos no contextuales.

Rank	Algoritmo	Tiempo
1	LinUCB	2:43:45
2	DynUCB	2:50:05
3	CLUB	2:54:11

(b) Algoritmos contextuales.

Tabla 5.4: Ranking de los algoritmos según el tiempo de ejecución en la base de datos *MovieLens*.

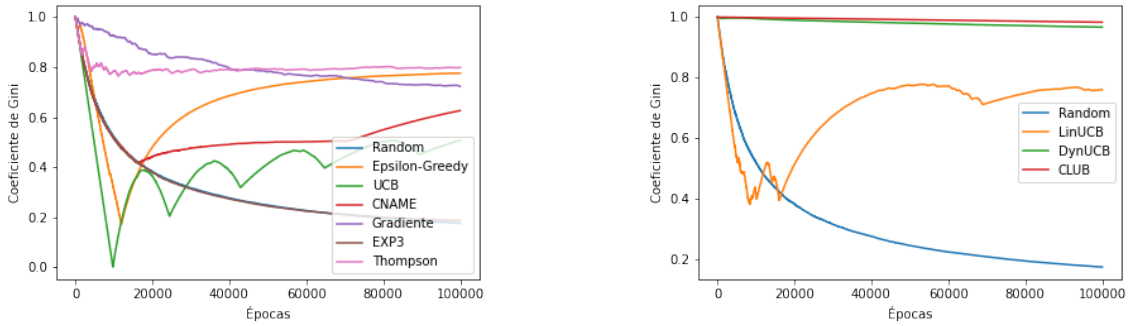
Como vemos, todos los algoritmos no contextuales tienen tiempos razonables excepto el algoritmo

Gradiente, que tarda casi el triple que el siguiente de la lista. Por el resto, los siguientes algoritmos son UCB y *Thompson Sampling*, que tiene sentido que sean que sigan al Gradiente: el primero tiene que computar todos los intervalos de confianza de las acciones en cada época, mientras que *Thompson Sampling* tiene que barajar para cada acción un número de una distribución beta distinta. Por otro lado, CNAME, ε-greedy y EXP3 no tienen ninguno de estos problemas a la hora de computar, por lo que son los más rápidos (aunque con poca diferencia).

Además, si algo se puede destacar son los altos tiempos que se gestionan en los algoritmos no contextuales debido a las costosas operaciones matriciales. Aunque LinUCB sea el mejor algoritmo en cuanto a *recall* acumulado se refiere, habría que valorar si vale la pena soportar unos tiempos de ejecución tan altos.

5.3. Análisis de la variedad en resultados

No solo interesa saber el *recall* acumulado. Como se ha visto en la Sección 2.4, hay otras propiedades importantes que tiene un buen algoritmo de recomendación, como la variedad de resultados. Para este objetivo, utilizaremos el coeficiente de Gini explicado en (2.1). De nuevo, se mostrarán los resultados de los distintos algoritmos optimizados en gráficas en la Figura 5.14.



(a) Algoritmos no contextuales.

(b) Algoritmos contextuales.

Figura 5.14: Comparativa del coeficiente de Gini tras 100.000 épocas en los algoritmos optimizados en *MovieLens*.

El problema que vemos en los algoritmos contextuales que no daban buenos resultados (CLUB y DynUCB) es que tienen los coeficientes de Gini más altos con diferencia. Por lo tanto, se están sobreespecializando y solo son capaces de recomendar unos pocos *items*. Por otro lado, vemos que aunque LinUCB daba los mejores resultados en *recall* acumulado, no muestra mucha variedad en sus recomendaciones.

Respecto a los algoritmos contextuales, Thompson Sampling y ε-greedy dan los peores resultados, el algoritmo Gradiente y CNAME mejoran un poco y UCB es el algoritmo que recomienda las opciones

más variadas. EXP3 tiene tan buenos resultados porque en sus 100.000 primeras épocas funciona prácticamente como el algoritmo aleatorio. De nuevo, para poder juzgarlo como al resto se adjuntan los resultados tras un millón de épocas en la Figura 5.15.

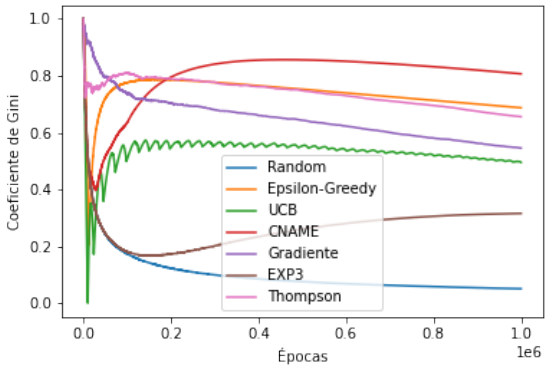


Figura 5.15: Coeficiente de Gini de algoritmos en la base de datos *MovieLens* tras un millón de épocas.

Como vemos, el problema de CNAME no se resuelve, sino que empeora (puede ser esa la causa de su bajada en *recall* acumulado). Por otro lado, tanto *Thompson Sampling* como ε-greedy han mejorado bastante. El algoritmo Gradiente y UCB muestran muy buenos resultados, pero sin duda el que más variedad ofrece (en detrimento del *recall* acumulado) es EXP3.

6

Conclusiones

En esta sección se resume todo el proyecto, recalcando los puntos más importantes y los objetivos cumplidos. También se quiere hacer hincapié en las contribuciones y novedades que este proyecto ha tenido y en las que no ha tenido, es decir, en aquellas que quedan como trabajo para otros proyectos más amplios.

6.1. Resumen y contribuciones

El tema central de este trabajo era crear una librería de bandidos multi-brazo para sistemas de recomendación que pudiera utilizarse en labores de investigación. Para ello, se han implementado 9 algoritmos distintos: ε-greedy, UCB, CNAME, Thompson Sampling, EXP3, Gradiente, LinUCB, DynUCB y CLUB. Entre ellos, se han incluído tanto algoritmos contextuales como no contextuales, buscando también establecer una comparativa entre los dos tipos. También se ha diseñado una herramienta *Analysis* que permite evaluar y comparar los algoritmos implementados con distintas métricas.

Para evaluar los resultados de estos algoritmos se han usado las bases de datos *MovieLens* y *CM100K*, aunque los algoritmos contextuales solo han podido evaluarse en *MovieLens*, ya que *CM100K* no incluía un equivalente a los géneros de películas de *MovieLens* que permitiera fabricar un contexto pequeño. Respecto a los resultados de los experimentos, lo más relevante es que los algoritmos contextuales más complejos (DynUCB y CLUB) funcionan mucho peor que el resto porque se dedican a sobreexplotar unos pocos *items*. Aunque esto pudiera ser porque el contexto debe ser más complejo, hay un algoritmo que funciona bien con este contexto: LinUCB, que da los mejores resultados en *recall* acumulado. Sin embargo, también tiende a sobreespecializarse y podría pasarle factura a más largo plazo. Además, debido a sus costosas operaciones matriciales estos algoritmos tardan mucho más en ejecutarse que los no contextuales (pueden llegar a un factor 50).

Respecto a los algoritmos no contextuales, uno de los resultados más importantes ha sido que EXP3 aprende muy lento, comportándose durante mucho tiempo igual que el algoritmo aleatorio. Sin embargo, una vez que funciona bien es el que más variedad ofrece en sus recomendaciones. Por otro lado, el algoritmo CNAME funciona bien a corto plazo, pero a partir de un número elevado de

épocas comienza a sobreespecializarse y a dar peores resultados que los otros algoritmos. Respecto al resto de algoritmos, todos dan una precisión parecida en sus recomendaciones, pero junto a EXP3 es UCB el que más variedad ofrece en sus recomendaciones. Por último, se quiere destacar que todos los algoritmos no contextuales tienen un coste muy bajo comparados con los contextuales, pero el Gradiente es especialmente costoso (aunque compensa en recomendaciones a corto plazo).

En conclusión, la contribución de este proyecto ha sido una librería funcional con unos algoritmos variados y con distintas ventajas. Esta librería permite investigar los algoritmos y da flexibilidad al usuario para modificarlos o extenderlos. Junto a ella, también se aporta en este documento información sobre todos los algoritmos; se espera que el lector la considere útil y enriquecedora.

6.2. Trabajo futuro

Aunque esta librería es completa, pretende ser más una puerta que abra muchas más, enseñado la cara de algoritmos de aprendizaje reforzado poco conocidos. Por ello, aunque los algoritmos se ha procurado que los algoritmos estén lo más optimizados que fuera posible, seguramente puedan estarlo mucho más. En especial los algoritmos contextuales implican tratar con matrices de grandes dimensiones y problemas de grafos en los que podría mejorarse la implementación. Como idea inicial se propone emigrar a lenguajes que puedan ser más eficientes, implementar esta librería sobre *TensorFlow* o paralelizar las operaciones matriciales en una GPU.

Además, los algoritmos que implicaban *clustering* (DynUCB y CLUB) mostraban resultados mucho peores que el resto de algoritmos. Quizá solo sirvan para otro tipo de problemas de recomendación distintos al de *MovieLens* (con la matriz de *ratings* más llena, por ejemplo), quizás necesitan más iteraciones para arrancar o quizá haya que utilizar otro tipo de contextos. Solamente investigando a fondo estos dos algoritmos se podría hacer otro trabajo, ya que en la literatura que se ofrece sobre estos algoritmos no se contempla este problema que se observa con *MovieLens*.

Otra faceta que queda por explorar es la de los problemas no estacionarios. Algunos de los algoritmos (como ε-greedy o gradiente) están preparados para afrontar problemas no estacionarios, pero las bases de datos necesarias son mucho más complejas: los *ratings* deben evolucionar con el tiempo y tiene que ser posible la introducción de nuevos *items*. Aunque supondría un completo cambio de vista, merece la pena explorar este área, ya que los problemas reales suelen ser no estacionarios.

Por último, también se propone explicar los algoritmos desde una perspectiva más teórica, indagando en los principios matemáticos que hay detrás de todos los algoritmos. Aunque se ha dedicado el Apéndice B a explicar los fundamentos en los que se basa el contexto y se ha dedicado un capítulo entero a explicar los algoritmos, no se ha hecho hincapié en la naturaleza matemática detrás de la mayoría de algoritmos. En especial, en algoritmos como CLUB, EXP3 o DynUCB no se deja muy claro por qué funcionan bien, sino que se ha intentado explicar únicamente su funcionamiento.

Bibliografía

[Aggarwal, 2016] Aggarwal, C. C. (2016). *Recommender Systems: The Textbook*. Springer.

[Cañamares and Castells, 2018] Cañamares, R. and Castells, P. (2018). Should i follow the crowd? a probabilistic analysis of the effectiveness of popularity in recommender systems. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '18, page 415–424, New York, NY, USA. Association for Computing Machinery.

[Cañamares et al., 2019] Cañamares, R., Redondo, M., and Castells, P. (2019). Multi-armed recommender system bandit ensembles. RecSys '19, page 432–436, New York, NY, USA. Association for Computing Machinery.

[Chapelle and Li, 2011] Chapelle, O. and Li, L. (2011). An empirical evaluation of thompson sampling. In Shawe-Taylor, J., Zemel, R., Bartlett, P., Pereira, F., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc. (Descargar).

[Gentile et al., 2014] Gentile, C., Li, S., and Zappella, G. (2014). Online clustering of bandits.

[Harper and Konstan, 2015] Harper, F. M. and Konstan, J. A. (2015). The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4).

[Li et al., 2010] Li, L., Chu, W., Langford, J., and Schapire, R. E. (2010). A contextual-bandit approach to personalized news article recommendation. *CoRR*, abs/1003.0146. (Descargar).

[Nguyen and Lauw,] Nguyen, T. T. and Lauw, H. W. Dynamic clustering of contextual multi-armed bandits. *Proceedings of the 2014 ACM International Conference on Information and Knowledge Management*, (November 3-7). (Descargar).

[Sanz-Cruzado et al., 2019] Sanz-Cruzado, J., Castells, P., and López, E. (2019). A simple multi-armed nearest-neighbor bandit for interactive recommendation. RecSys '19, page 358–362, New York, NY, USA. Association for Computing Machinery.

[Seldin et al., 2013] Seldin, Y., Szepesvári, C., Auer, P., and Abbasi-Yadkori, Y. (2013). Evaluation and analysis of the performance of the exp3 algorithm in stochastic environments. In Deisenroth, M. P., Szepesvári, C., and Peters, J., editors, *Proceedings of the Tenth European Workshop on Reinforcement Learning*, volume 24 of *Proceedings of Machine Learning Research*, pages 103–116, Edinburgh, Scotland. PMLR. (Descargar).

[StatsDirect, 2013] StatsDirect (2013). Statsdirect statistical software. (Ver aquí).

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA.

[Vargas, 2014] Vargas, S. (2014). Novelty and diversity enhancement and evaluation in recommender systems and information retrieval.

[Walsh et al., 2012] Walsh, T., Szita, I., Diuk, C., and Littman, M. (2012). Exploring compact reinforcement-learning representations with linear regression.

[Zhang et al., 2018] Zhang, X., Zhou, Q., He, T., and Liang, B. (2018). Con-cname: A contextual multi-armed bandit algorithm for personalized recommendations. In Kůrková, V., Manolopoulos, Y., Hammer, B., Iliadis, L., and Maglogiannis, I., editors, *Artificial Neural Networks and Machine Learning – ICANN 2018*, pages 326–336, Cham. Springer International Publishing. (Descargar).

Apéndices

A

Algoritmo Tf-idf

Este algoritmo se utiliza para calcular el vector contexto de una cierta base de datos. En ella, se espera que haya dos columnas: la primera con un cierto *item* y la segunda con un cierto *tag*, indicando que dicho *item* tiene ese *tag* asociado. El algoritmo consta de dos partes: el cálculo del *Tf* y el cálculo del *idf* para cada *tag*.

Comencemos por el *tf* (*Term frequency*), donde *term* es, en este caso, un cierto *tag*. [1] En primer lugar, denominemos la frecuencia de un cierto *tag* t en un cierto *item* $i \in \mathcal{I}$ como $f_{t,i}$. Entonces, el *tf* de un cierto *tag* en un cierto *item* es:

$$tf(t,i) = \frac{f_{t,i}}{\sum_{j \in \mathcal{I}} f_{t,j}}. \tag{A.1}$$

Ahora, veamos el *idf* (*inverse document frecuency*) de cada *tag* t. Este término nos da una idea de cuánta información da dicho *tag* (por ejemplo, si se encuentra a menudo en muchos *items* no da mucha información). Dicho esto, la fórmula que lo define es:

$$idf(t) = \frac{|\mathcal{I}|}{|\{i \in \mathcal{I} : f_{t,i} > 0\}|}, \tag{A.2}$$

donde $|\mathcal{I}|$ es el número de *items* de la base de datos y $|\{i \in \mathcal{I} : f_{t,i} > 0\}|$ es el número de *items* que contienen el *tag* en cuestión.

Finalmente, el *tf-idf* de un cierto *tag* t en un cierto *item* i se calcula mediante:

$$tfidf(t,i) = tf(t,i) \cdot idf(t). \tag{A.3}$$

Para formar el vector *contexto* de un cierto *item*, se obtiene el *tf-idf* de dicho *item* con cada *tag* y se agrupan todos los resultados. De esta forma, obtenemos un vector cuya dimensión es el número de

[1] Este algoritmo es utilizado habitualmente para evaluar palabras en un documento. Por ello, *term* se refiere a los *tags* y *document* se refiere a cada *item*, que tiene unos ciertos tags.

tags que haya en la base de datos.

Es importante que **el número de tags no sea muy grande** (no superior a 30), ya que esta va a ser la dimensión de más vectores y matrices en los algoritmos, y es necesario hacer multiplicaciones y matrices inversas con estas dimensiones. Por lo tanto, si el vector fuera demasiado grande podría hacer al algoritmo muy ineficiente.

Además, como ya se habrá podido observar, el contexto no cambia con el tiempo en esta librería. Si unimos esto a lo dicho anteriormente sobre la dimensionalidad del vector contexto, conviene que los $tags$ sean categorías en las que se puedan clasificar los *items* (por ejemplo, géneros en el caso de las películas).

B

Fundamentos matemáticos tras el contexto

Viendo los algoritmos contextuales de este trabajo podemos pensar qué tienen que ver las matrices M y los vectores b que se asocian a usuarios o *clusters*. En este apéndice se da respuesta a esas preguntas y se pretende dar una intuición sobre el sentido que tienen estas variables, basándonos en parte del artículo [Li et al., 2010].

Nuestro problema consiste en que queremos predecir qué recompensa R_t nos puede dar un brazo $a \in \mathcal{A}$ cuyo contexto es un vector de dimensión d llamado x_a. Por lo tanto, dada esta información nos gustaría tener otro vector de dimensión d llamado $\boldsymbol{\theta}_a^*$ tal que:

$$\boxed{\mathbf{E}(R_t|x_a) = x_a^T \boldsymbol{\theta}_a^*.} \tag{B.1}$$

Sin embargo, no disponemos de tal $\boldsymbol{\theta}_a^*$, por lo que hallaremos una aproximación que llamaremos $\boldsymbol{\theta}_a$. Si el algoritmo ha sido elegido un número m de veces, declaremos dos variables:

- D_a es una matriz de dimensión $m \times d$, donde las filas corresponden al contexto del *item* a en ese momento. Como en nuestro modelo el contexto no cambia, todas las filas de la matriz serán iguales.
- c_a es un vector de dimensión m cuyos valores son las recompensas obtenidas.

Tenemos una matriz con cada columna representando un parámetro del contexto y una serie de predicciones para cada contexto. En estos casos, podríamos hacer regresión linal y estimar $\boldsymbol{\theta}_a$ mediante mímimos cuadrados. Sin embargo, cuando los parámetros implicados están fuertemente correlados se utiliza la *ridge regression*:

$$\boxed{\boldsymbol{\theta}_a = (D_a^T D_a + I_d)^{-1} D_a^T c_a,} \tag{B.2}$$

donde I_d es la matriz identidad de dimensión d. Suponiendo que cada componente de c_a está condicionado únicamente a su correspondiente fila en D_a, se puede probar [Walsh et al., 2012] que, con una confianza de $1 - \delta$ podemos acotar nuestro máximo desvío:

$$|x_a^T \boldsymbol{\theta}_a - \mathbf{E}(R_t|x_a)| \leq \alpha\sqrt{x_a^T(D_a^T D_a + I_d)^{-1}x_a}, \tag{B.3}$$

donde $\alpha = 1 + \sqrt{\frac{\ln\frac{2}{\delta}}{2}}$. Por lo tanto, ya tenemos una cota superior que podemos fijar con cierta confianza. De esta forma, podemos aprovechar esta confianza para fabricar modelos UCB, tomando siempre el valor más optimista permitido por la cota.

Las matrices M_a y los vectores b_a no son más que una simplificación de estos cálculos:

$$M_a = D_a^T D_a + I_d, \tag{B.4a}$$

$$b_a = D_a^T c_a. \tag{B.4b}$$

C

Pseudocódigo de los algoritmos contextuales

En este apéndice se recogen los pseudocódigos de aquellos algoritmos que eran demasiado extensos y complejos como para introducirlos en el capítulo de algoritmia. Dichos algoritmos son CNAME, LinUCB, DynUCB y CLUB.

input : Acciones $\mathcal{A}$, usuarios $\mathcal{U}$, valor inicial $w \in (0, \infty)$.
output: Recompensa de cada época

for $a \in \mathcal{A}$ **do**
    $N(a) = 0$; /* Número de épocas */
    $Q(a) = 0$; /* estimación de recompensa de cada brazo */
end
for $t = 1 \dots T$ **do**
    $U_t \leftarrow$ recibir_usuario($\mathcal{U}$);
    $\mathcal{A}^* \leftarrow$ acciones_no_exploradas(U_t);
    $m \leftarrow N(arg\,\text{mín}_{a \in \mathcal{A}^*} Q(a))$;
    $p \leftarrow \frac{w}{w+m^2}$;
    $x \leftarrow random(0, 1)$;
    if $x > p$ **then**
        $A_t \leftarrow arg\,\text{máx}_{a \in \mathcal{A}^*} Q(a)$;
    end
    else
        $A_t \leftarrow$ eleccion_random($\mathcal{A}^*$);
    end
    $R_t \leftarrow$ obtener_recompensa(A_t);
    $N(A_t) \leftarrow N(A_t) + 1$;
    $Q(A_t) \leftarrow Q(A_t) + \frac{1}{N(A_t)}(R_t - Q(A_t))$;
end

Algoritmo C.1: Algoritmo CNAME.

input : Acciones $\mathcal{A}$, usuarios $\mathcal{U}$, valor inicial $\alpha \in \mathbb{R}^+$.
output: Recompensa de cada época

for $a \in \mathcal{A}$ **do**
  $M_a \leftarrow \mathbf{I}_{d \times d}$;
  $b_a \leftarrow \mathbf{0}_d$;
end
for $t = 1 \dots T$ **do**
  $U_t \leftarrow$ recibir_usuario($\mathcal{U}$);
  $\mathcal{A}^* \leftarrow$ acciones_no_exploradas(U_t);
  for $a \in \mathcal{A}^*$ **do**
    $x_a \leftarrow$ vector_contextos(a);
    $\theta_a \leftarrow M_a^{-1} b_a$;
    $p_a \leftarrow \theta_a^T x_a + \alpha \sqrt{x_a^T M_a^{-1} x_a}$;
  end
  $a_t \leftarrow arg\,\text{máx}_{a \in \mathcal{A}^*} p_a$; /* Elección de brazo */
  $R_t \leftarrow$ obtener_recompensa(a_t);
  $M_a \leftarrow M_a + x_a x_a^T$;
  $b_a \leftarrow b_a + R_t x_a$;
end

Algoritmo C.2: Algoritmo LinUCB.

input : Acciones $\mathcal{A}$, usuarios $\mathcal{U}$, número de *clusters* $k \in \mathbb{N}$.
output: Recompensa de cada época

for $u \in \mathcal{U}$ **do**
  $M_u \leftarrow \mathbf{I}_{d\times d}$;
  $b_u \leftarrow \mathbf{0}_d$;
  asignacion_aleatoria_cluster(u);
end
for $k = 1 \cdots K$ **do**
  $M_k \leftarrow \mathbf{I}_{d\times d} + \sum_{u\in C_k}(M_u - \mathbf{I}_{d\times d})$;
  $b_k \leftarrow \sum_{u\in C_k} b_u$;
end
for $t = 1 \ldots T$ **do**
  $U_t \leftarrow$ recibir_usuario($\mathcal{U}$);
  $C_k \leftarrow$ pertenece_cluster(U_t);
  $\mathcal{A}^* \leftarrow$ acciones_no_exploradas(U_t);
  for $a \in \mathcal{A}^*$ **do**
    $x_a \leftarrow$ vector_contextos(a) ;
    $\theta_a \leftarrow M_k^{-1} b_k$;
    $p_a \leftarrow \theta_a^T x_a + \alpha\sqrt{x_a^T M_k^{-1} x_a \log(t+1)}$;
  end
  $A_t \leftarrow arg\,\text{máx}_{a\in\mathcal{A}^*}\, p_a$; /* Elección de brazo */
  $R_t \leftarrow$ obtener_recompensa(a_t);
  $M_{U_t} \leftarrow M_{U_t} + x_a x_a^T$; /* Actualización del usuario */
  $b_{U_t} \leftarrow b_{U_t} + R_t x_a$;
  $k' \leftarrow arg\,\text{mín}_{k'\in\{1\ldots K\}}\ ||M_{U_t}^{-1} b_{U_t} - M_{k'}^{-1} b_{k'}||$; /* Asignación al cluster más cercano */
  if $k \neq k'$ **then**
    mover_usuario($U_t, C_{k'}$);
    recalcula_M(C_k,$C_{k'}$) ; /* Se deben recomputar M,b como en 7-8 */
    recalcula_b(C_k,$C_{k'}$);
  end
end

Algoritmo C.3: Algoritmo DynUCB.

input : Acciones $\mathcal{A}$, usuarios $\mathcal{U}$, parámetro de exploración $\alpha > 0$, parámetro de eliminación $\beta > 0$.
output: Recompensa de cada época

for $u \in \mathcal{U}$ **do**
  $M_u \leftarrow \mathbf{I}_{d\times d}$;
  $b_u \leftarrow \mathbf{0}_d$;
  asignacion_aleatoria_cluster(u);
end
$G \leftarrow$ obtener_grafo_completo($\mathcal{U}$);
$C_1 \leftarrow G$ (C_1 es el único *cluster* en tiempo 0);
for $t = 1 \dots T$ **do**
  $U_t \leftarrow$ recibir_usuario($\mathcal{U}$);
  $C_k \leftarrow$ pertenece_cluster(U_t,$\mathbf{C}_t$);
  $\mathcal{A}^* \leftarrow$ acciones_no_exploradas(U_t);
  for $a \in \mathcal{A}^*$ **do**
    $x_a \leftarrow$ vector_contextos(a) ;
    $\theta_a \leftarrow M_k^{-1} b_k$;
    $p_a \leftarrow \theta_a^T x_a + \alpha\sqrt{x_a^T M_k^{-1} x_a \log(t+1)}$;
  end
  $A_t \leftarrow arg\,\text{máx}_{a\in\mathcal{A}^*}\, p_a$; /* Elección de brazo */
  $R_t \leftarrow$ obtener_recompensa(a_t);
  $M_{U_t} \leftarrow M_{U_t} + x_a x_a^T$; /* Actualización del usuario */
  $b_{U_t} \leftarrow b_{U_t} + R_t x_a$;
  /* Revisión de las aristas de U_t */
  $CB_{U_t} \leftarrow \beta\sqrt{\frac{1+\log(1+T_{U_t})}{1+T_{U_t}}}$;
  for $U' \in usuarios\_conectados(U_T)$ **do**
    $CB_{U'} \leftarrow \beta\sqrt{\frac{1+\log(1+T_{U'})}{1+T_{U'}}}$;
    if $||M_{U_T}^{-1} b_{U_T} - M_{U'}^{-1} b_{U'}|| > CB_{U_T} + CB_{U'}$ **then**
      elimina_arista(G, U_t, U');
    end
  end
  $\mathbf{C}_t \leftarrow$ get_componentes_conexas(G);
end

Algoritmo C.4: Algoritmo CLUB.

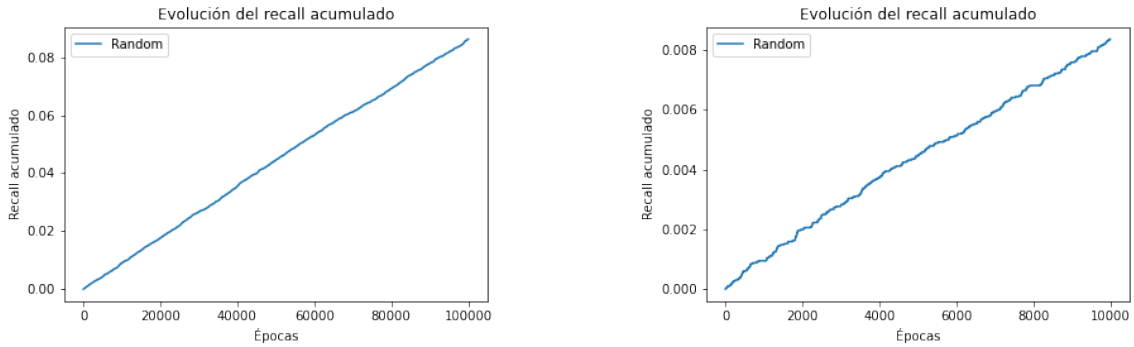
D

Análisis de algoritmos en CM100K

Los resultados obtenidos en esta base de datos no incluye los algoritmos contextuales, ya que no existe nada que se pueda usar como un contexto de bajas dimensiones. A continuación se muestra el mismo análisis que se hizo para la base de datos MovieLens: búsqueda en rejilla intentando maximizar el *recall* acumulado y posterior análisis del *recall* acumulado, tiempo de ejecución y coeficiente de Gini para los algoritmos.

D.1. Análisis del *recall* acumulado

En primer lugar, se buscarán los mejores hiperparámetros en cada algoritmo para esta base de datos. Después, se evaluará como de bueno es cada algoritmo. Para tener una referencia, veamos cómo funciona el recomendador aleatorio tras 100.000 épocas y un millón de épocas:



(a) 100.000 épocas.

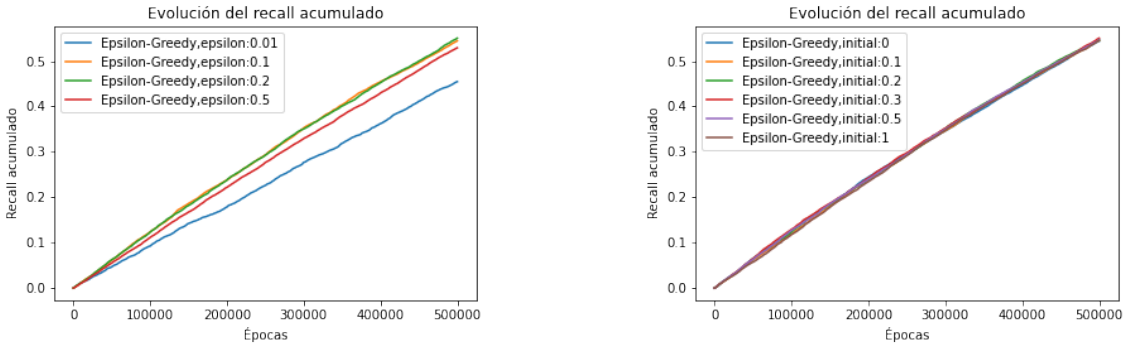
(b) Un millón de épocas.

Figura D.1: *Recall* acumulado del algoritmo aleatorio en *CM100K*

Como vemos, tras un millón de épocas se han descubierto casi todas las buenas recomendaciones: un $90\,\%$. Por ello, no interesa mirar tan a largo plazo qué pasa, ya que la mayoría de algoritmos conseguirán más de un $90\,\%$, pero en el resultado final no se distinguirán mucho. Por ello, en esta base de datos se mirará qué pasa tras 100.000 épocas y tras 500.000 épocas, ya que interesa más ver cómo se comportan en momentos intermedios como este.

D.1.1. Búsqueda en rejilla de hiperparámetros

Comencemos con el algoritmo ε**-greedy**. De nuevo, estamos antes un problema estacionario, por lo que los únicos parámetros que debemos evaluar son el coeficiente de exploración y el valor inicial de la recompensa esperada cuando no se tiene información. Son independientes, ya que el valor inicial solo influye en el arranque en frío. En la Figura D.2 se muestran los dos hiperparámetros tras 500.000 épocas.



(a) Análisis de ε.

(b) Análisis de recompensa inicial.

Figura D.2: *Recall* acumulado de ε-greedy tras 500.000 épocas en la base de datos *CM100K*.

Como vemos, el mejor valor para ε sigue siendo $0{,}2$, mientras que el valor inicial no nota ninguna diferencia en tantas épocas. Si hacemos *zoom* en las primeras 100.000 épocas (Figura D.3) podemos ver que sigue sin haber diferencias prácticamente. Por lo tanto, da igual tener una inicialización optimista o pesimista en este modelo.

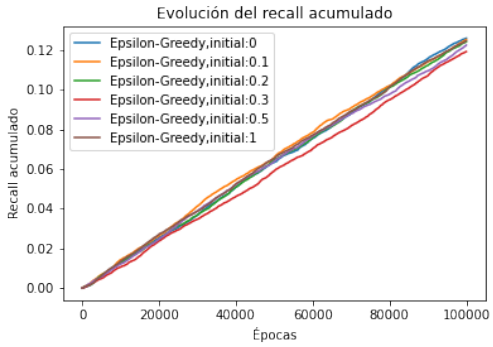
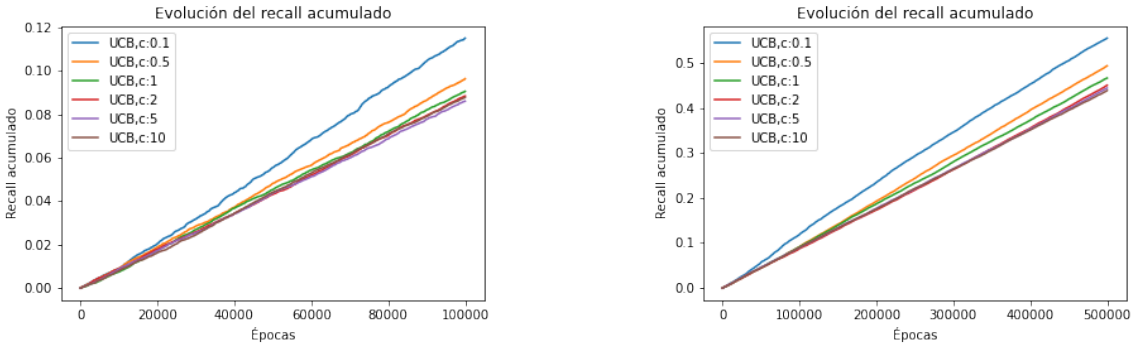


Figura D.3: *Recall* acumulado de ε-greedy tras 100.000 épocas en la base de datos *CM100K*.

Pasemos al algoritmo **UCB**. De nuevo, al estar en un problema estacionario solo queda fijar el parámetro $c > 0$, es decir, el margen de confianza. En la Figura D.4 se muestra la evolución tras 100.000 y 500.000 épocas.

Como puede verse, el valor más bajo $c = 0{,}1$ vuelve a ser el que da mejores resultados, tanto a



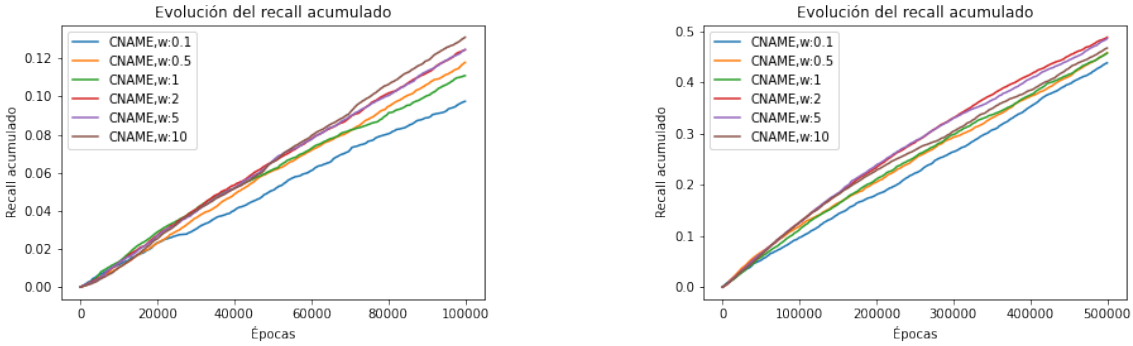
(a) 100.000 épocas.

(b) 500.000 épocas.

Figura D.4: *Recall* acumulado de UCB en la base de datos *CM100K*.

corto como a largo plazo (aunque a corto plazo se nota aún más).

Ahora veamos el algoritmo **CNAME**, que sintetiza los dos algoritmos anteriores. En él, solo se analiza el parámetro de exploración $w > 0$, mostrado en la Figura D.5.



(a) 100.000 épocas.

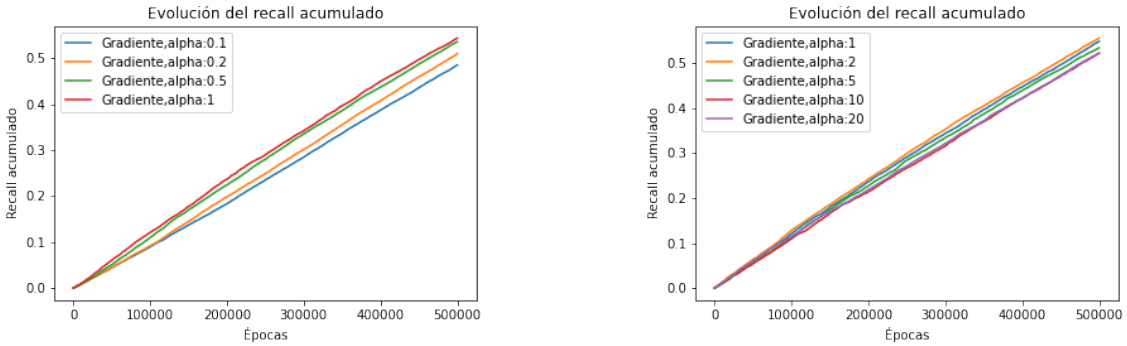
(b) 500.000 épocas.

Figura D.5: *Recall* acumulado de CNAME en la base de datos *CM100K*.

Puede observarse que en un número bajo de épocas beneficia tener un coeficiente de exploración alto, como $w = 10$. Sin embargo, tras ejecutar 500.000 épocas estos parámetros altos pierden fuerza frente a otros con un coeficiente más moderado, como $w = 2$.

Veamos ahora el algoritmo **Gradiente**. Como de nuevo el problema es estacionario solo tenemos que fijar la constante de aprendizaje $\alpha > 0$. En la Figura D.6 se muestran los resultados con distintos valores para 500.000 épocas.

Se puede ver cómo se alcanza un máximo en $\alpha = 2$, mientras que con valores más altos o más bajos decrece el *recall* acumulado, por lo que este es el valor elegido en este caso. Sin embargo, se observa que los valores bajos perjudican bastante más que los valores altos, exactamente el mismo



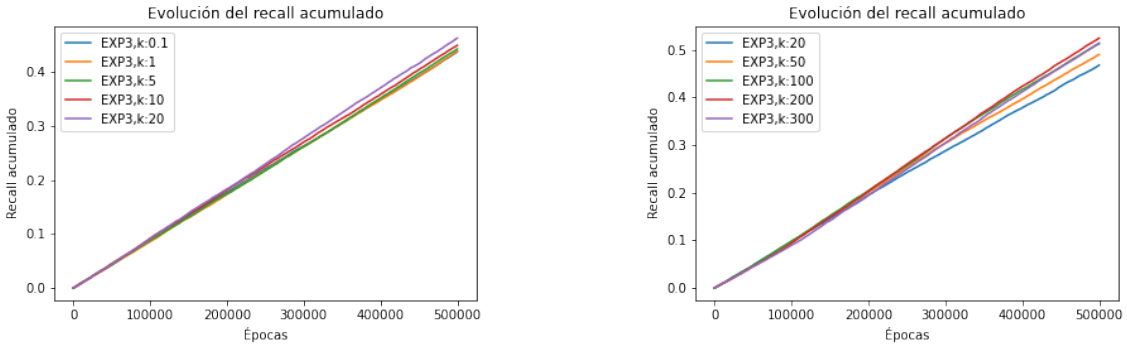
(a) Para valores α bajos.

(b) Para valores α altos.

Figura D.6: *Recall* acumulado del algoritmo Gradiente en la base de datos *CM100K*.

comportamiento que se observaba en la base de datos MovieLens.

Pasemos a ver el algoritmo **EXP3**. En él, solo hay que analizar el valor $k > 0$ que define cómo de lento decrece la tasa de aprendizaje. Una vez dicho esto, se adjunta el análisis habitual en la Figura D.7.



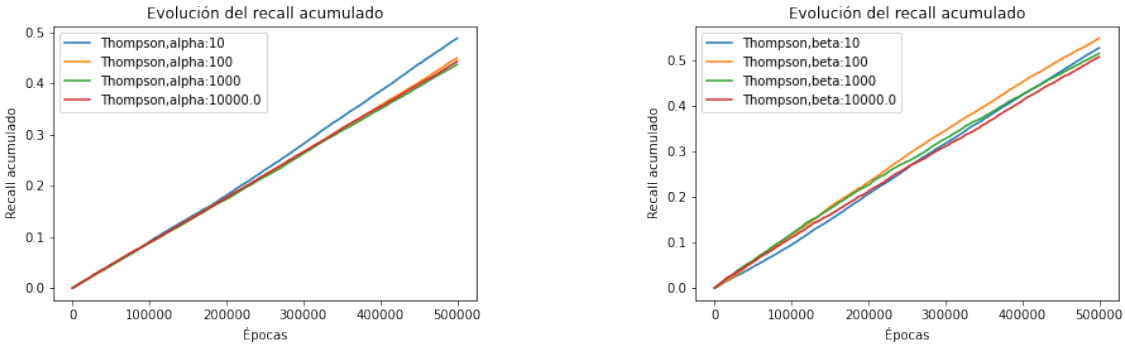
(a) Para valores k bajos.

(b) Para valores k altos.

Figura D.7: *Recall* acumulado de EXP3 en la base de datos *CM100K*.

Este algoritmo tiene el problema habitual de EXP3: tarda demasiado en alcanzar buenos resultados. Como esta base de datos es bastante más pequeña que EXP3, solo podemos empezar a notar un poco cómo se distancia del recomendador aleatorio. De todas formas, parece que le benefician valores de k especialmente altos, como 200.

Por último, veremos el algorimto **Thompson Sampling**. En él, hay que analizar si es mejor una inicialización optimista con $\alpha \gg \beta$ o una inicialización pesimista con $\beta \gg \alpha$. Para ello, en las inicializaciones pesimistas se mantendrá $\alpha = 1$ y en las pesimistas $\beta = 1$. Se adjuntan resultados en la Figura D.8.



(a) Inicialización optimista ($\beta = 1$).

(b) Inicialización pesimista ($\alpha = 1$).

Figura D.8: *Recall* acumulado de *Thompsona Sampling* en la base de datos *CM100K*.

Como puede verse, de nuevo es mejor una incialización pesimista. En concreto, ahora dan mejores resultados valores bajos para β, como 100. Sin embargo, cualquier valor que suponga una inicialización pesimista es mejor que una inicialización optimista.

D.1.2. Comparativa de algoritmos

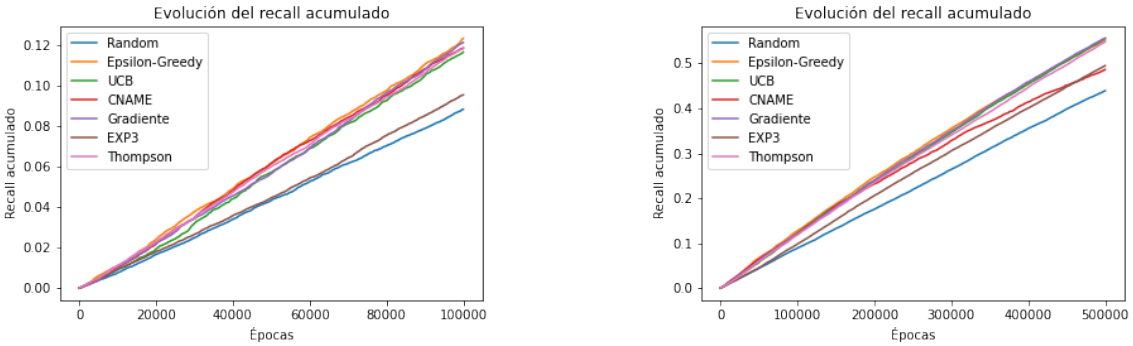
Con los resultados anteriores, los hiperparámetros escogidos son los que se muestran en la Tabla D.1. De todas formas, los resultados obtenidos son bastante más plano, ya que se nota una mejoría mucho menor que en la base de datos MovieLens. Esto probablemente se deba a la pequeña dimensión de la base de datos, pero también proporciona nueva información: muestra cómo funcionan estos algoritmos con bases de datos más pequeñas y con mayor densidad de datos.

Algoritmo	Selección
ε-greedy	$\varepsilon = 0{,}2$; $initial = 0$
UCB	$c = 0{,}1$
CNAME	$w = 2$
Gradiente	$\alpha = 2$
EXP3	$k = 200$
Thompson Sampling	$\alpha = 1$; $\beta = 100$

Tabla D.1: Tabla de selección de hiperparámetros.

Ahora, veremos cómo evoluciona el *recall* acumulado tras 100.000 épocas y tras 500.000 épocas con cada uno de los algoritmos en la Figura D.9.

En primer lugar, los resultados son bastante planos para 100.000 épocas: aunque todos son similares, muestran mejores resultados que el recomendador aleatorio. Un resultado distinto es, de nuevo, EXP3, que se comporta muy parecido al algoritmo aleatorio.



(a) 100.000 épocas.

(b) 500.000 épocas.

Figura D.9: Análisis del *recall* acumulado para los mejores algoritmos.

En cambio, tras 500.000 épocas se ve un cambio más interesante: CNAME parece escaparse por abajo y empeora sus resultados respecto al resto, mientras que EXP3 va mejorando. Los demás algoritmos siguen comportándose de manera similar, con un *recall* acumulado parecido. En general, los resultados parecen más plano que en *MovieLens*, posiblemente porque se está tratando con una base de datos más pequeña.

D.2. Análisis de los tiempos de ejecución

Ahora, veremos los tiempos de ejecución de los algoritmos. No sólo nos permitirá saber cuáles son los algoritmos más costosos, sino que también nos permitirá saber si los tiempos de ejecución son lineales respecto al número de épocas. Se adjuntan en la Figura D.10 y en la Tabla D.2.

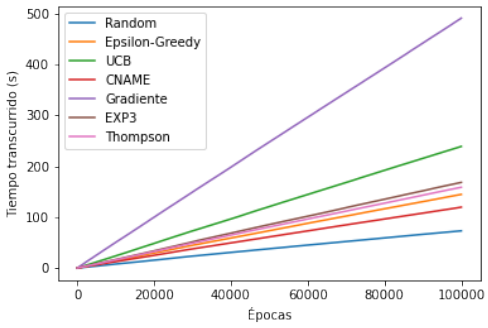


Figura D.10: Análisis de los tiempos de ejecución para los mejores algoritmos tras 100.000 épocas.

Aunque los resultados han cambiado un poco respecto a la base de datos *MovieLens* (ahora son más bajos), los resultados siguen siendo los mismos: el algoritmo gradiente de lejos es el más costoso, aunque no es tanta la proporción respecto al siguiente (ha pasado de casi el triple a el doble). Además, el algoritmo CNAME funciona especialmente rápido: como las ejecuciones son lineales, este algoritmo

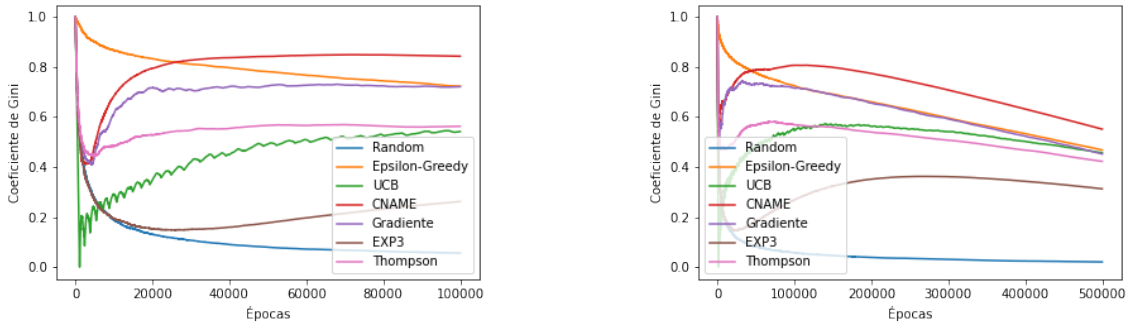
Rank	Algoritmo	*Recall* ac.
1	CNAME	0:01:59
2	ε-greedy	0:02:24
3	*Thompson*	0:02:38
4	EXP3	0:02:48
5	UCB	0:03:59
6	Gradiente	0:08:11

Tabla D.2: Ranking del tiempo de ejecución en *CM100K* para 100.000 épocas.

solo tardaría 20 minutos en ejecutarse, frente a los 80 del algoritmos gradiente.

D.3. Análisis del coeficiente de Gini

Por último, haremos un análisis sobre la variedad de recomendaciones que hace cada algoritmo. Como hemos dicho, las gráficas de *recall* son muy planas, por lo que en este caso el coeficiente de Gini puede ayudar a determinar cuáles son los mejores algoritmos. A continuación se adjuntan los resultados para 100.000 épocas y 500.000 épocas en la Figura D.11.



(a) 100.000 épocas.

(b) 500.000 épocas.

Figura D.11: Análisis del coeficiente de Gini para los mejores algoritmos.

En primer lugar, hay un detalle muy sorprendente tras 500.000 épocas: todos los algoritmos reducen su coeficiente de Gini tras muchas épocas, por lo que ninguno se sobreespecializa demasiado en *CM100K*. Además, de nuevo parece que el problema de CNAME se encuentra en su alto coeficiente de Gini, mientras que EXP3 es mejor en cuanto a recomendaciones variadas con el mismo *recall* acumulado que EXP3.